

**VERIFICATION AND ANALYSIS OF  
WEB SERVICE COMPOSITION**

**TAN TIAN HUAT**

**NATIONAL UNIVERSITY OF SINGAPORE**

2013

**VERIFICATION AND ANALYSIS OF WEB SERVICE  
COMPOSITION**

TAN TIAN HUAT

(B.Sc. (Hons.), National University of Singapore, 2009)

A THESIS SUBMITTED FOR THE DEGREE OF

**DOCTOR OF PHILOSOPHY**

DEPARTMENT OF COMPUTER SCIENCE

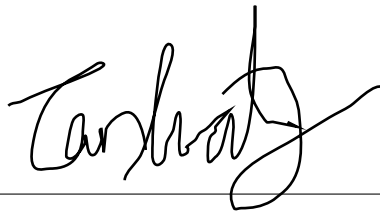
NUS GRADUATE SCHOOL FOR INTEGRATIVE SCIENCES AND  
ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2013

## Declaration

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

A handwritten signature in black ink, appearing to read 'Tan Tian Huat', is positioned above a horizontal line.

Tan Tian Huat  
21 Aug 2013

## Acknowledgements

It would be not possible to complete my thesis without the encouragement and help of people around me, who give me valuable instructions and assistance during the whole of my Ph.D. journey.

First and foremost, I would like to give my deepest and heartfelt gratitude to my supervisor, Professor Dong Jin Song, for his stimulating guidance, advice and encouragement during these past four years. Professor Dong is a very caring professor and I am deeply impressed by his good personality since I met him in his class. During the PhD candidature, he gives me great amount of freedom to pursue the research direction that excited me, and at the same time, he is constantly guiding me towards the right direction in doing research.

I am deeply grateful to my mentors Dr. Sun Jun and Dr. Liu Yang, who act like friends and co-supervisors in the past four years. I thank them for introducing me to the exciting area of web service composition verification. Their supervision and insightful suggestions on research have triggered me many interesting ideas and nourished my intellectual maturity that I will benefit for my whole life. My sincere appreciation also goes to Dr. Étienne André for his involvement and crucial contribution.

I would like to thank my seniors Dr. Chen Chunqing, Dr. Zhang Xian, Dr. Zhang Shaojie, Dr. Zheng Manchun, fellow students Song Songzheng, Liu Yan, Shi Ling, and all the juniors for your support and friendships through my Ph.D. study. And I am grateful to all my colleagues and friends in PAT group, NUS and elsewhere for their support and encouragement throughout, some of whom have already been named.

Lastly, I wish to thank sincerely and deeply my parents for their encouragement, support, unconditional love and care. I would not have made it this far without them.

# Contents

<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Algorithms</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Summary of this thesis . . . . .	2
1.2 Overall Picture . . . . .	5
1.3 Thesis Outline . . . . .	6
1.4 Acknowledgement of Published Work . . . . .	7
<b>2 Background</b>	<b>9</b>
2.1 SOA and Web Service Composition . . . . .	9
2.1.1 SOA and Web Service . . . . .	9
2.1.2 Web Service Composition . . . . .	11
2.2 Basics of Model Checking . . . . .	13
2.3 System Modeling . . . . .	14
2.4 Specification and Verification . . . . .	15
2.4.1 Safety Property . . . . .	15
2.4.2 Liveness Property . . . . .	16

<b>3</b>	<b>Conformance Checking of Service Composition</b>	<b>19</b>
3.1	Modeling . . . . .	22
3.1.1	Choreography: Syntax and Semantics . . . . .	23
3.1.2	Orchestration: Syntax and Semantics . . . . .	27
3.2	Verification . . . . .	29
3.3	Prototype Synthesis . . . . .	31
3.4	Implementation and Evaluation . . . . .	37
3.5	Related Work . . . . .	39
<b>4</b>	<b>Verification with Compositional Partial Order Reduction</b>	<b>43</b>
4.1	Orchestration Language Orc . . . . .	45
4.1.1	Syntax . . . . .	45
4.1.2	Semantics . . . . .	49
4.1.3	Hierarchical Concurrent Processes (HCP) . . . . .	51
4.2	Compositional Partial Order Reduction (CPOR) . . . . .	54
4.2.1	Classic POR and CPOR . . . . .	56
4.2.2	CPOR Algorithm . . . . .	57
4.2.3	Soundness . . . . .	60
4.3	Evaluation . . . . .	62
4.4	Related work . . . . .	64
<b>5</b>	<b>Integrated Verification of Service Composition</b>	<b>67</b>
5.1	Motivating Example . . . . .	70
5.1.1	Computer Purchasing Services (CPS) . . . . .	71
5.1.2	BPEL Notations . . . . .	72
5.2	QoS-Aware Compositional Model . . . . .	73
5.2.1	QoS Attributes . . . . .	73

5.2.2	QoS for Composite Services . . . . .	74
5.2.3	Labeled Transition Systems . . . . .	75
5.3	Verification of Functional and Non-Functional Requirements . . . . .	78
5.3.1	Verification of Functional Requirement . . . . .	78
5.3.2	Integration of Non-Functional Requirement . . . . .	79
5.3.3	Discussion . . . . .	84
5.4	Evaluation . . . . .	84
5.4.1	Computer Purchasing Service (CPS) . . . . .	84
5.4.2	Loan Service (LS) . . . . .	85
5.4.3	Travel Agency Service (TAS) . . . . .	86
5.5	Related Work . . . . .	87
<b>6</b>	<b>Dynamic Synthesis of Response Time Requirement</b>	<b>89</b>
6.1	A Timed BPEL Example . . . . .	92
6.1.1	Vehicle Booking Service . . . . .	92
6.1.2	BPEL Notations . . . . .	93
6.2	Formal Model for Parametric Analysis . . . . .	94
6.2.1	Clocks, Parameters, and Constraints . . . . .	95
6.2.2	Syntax of Composite Services . . . . .	96
6.2.3	Semantic Models . . . . .	97
6.3	Dynamic Analysis with LTS . . . . .	99
6.3.1	Clock Activation . . . . .	100
6.3.2	Idling Function . . . . .	101
6.3.3	Bad Activity . . . . .	101
6.3.4	Operational Semantics . . . . .	102
6.3.5	State Space Exploration . . . . .	103

6.3.6	Application to an Example . . . . .	103
6.4	Local Time Requirement Synthesis . . . . .	105
6.4.1	Synthesis of Local Time Requirement . . . . .	105
6.4.2	Addressing the Bad States . . . . .	107
6.4.3	Synthesis Algorithms . . . . .	107
6.4.4	Application to the Running Example . . . . .	109
6.4.5	Soundness . . . . .	110
6.5	Evaluation . . . . .	113
6.5.1	Stock Market Indices Service . . . . .	113
6.5.2	Computer Purchasing Services . . . . .	114
6.5.3	Travel Booking Service . . . . .	115
6.6	Related Work . . . . .	116
<b>7</b>	<b>Conclusion</b>	<b>119</b>
7.1	Summary . . . . .	119
7.2	Ongoing and Future Work . . . . .	121
	<b>Bibliography</b>	<b>123</b>



## Summary

A Web service is a self-describing, self-contained autonomous software system available via a network, such as the Internet. A Web service is dedicated for a business task, such as the booking of air ticket. Web service composition is to make use of existing heterogeneous services on the Web as components to achieve a business goal. By reusing the existing Web services, one can reduce the development time, and at the same time, increase the reliability of the service after composition. Our work is focused on verification and analysis of Web service composition.

In recent years, many Web service composition languages have been proposed. There are two different viewpoints of these Web service composition languages, namely Web service choreography and Web service orchestration. Web service choreography describes collaboration protocols of cooperating Web service participants from a global view. Web service orchestration describes collaboration of the web services in predefined partners. In order to link these two different views, we present model-based methods for automatic analysis of Web service compositions. We verify whether designs from two different views are consistent or not, by refinement checking with specialized optimizations. If these two views do not match, we also propose repair mechanism to address the problem.

Subsequently, we focus on the verification of Web service composition from the perspective of Web service orchestration. A challenge to verify Web service composition is that, the highly concurrent nature of Web service orchestration has introduced the state-explosion problem to search-based verification methods like model checking. To address the state-explosion problem, we present a new method, called Compositional Partial Order Reduc-

tion (CPOR) for verification of Web service orchestration. CPOR aims to provide greater state-space reduction than classic partial order reduction methods in the context of hierarchical concurrent processes.

Non-functional requirement, such as response time requirement, are important to Web service composition. To integrate non-functional requirements as part of the verification process, we further propose an automated approach to verify combined functional and non-functional requirements directly based on the semantics of web service composition. Model checking algorithms are developed to verify safety properties and liveness properties, in the forms of state reachability checking and Linear Temporal Logic (LTL) checking.

Response time requirement is often provided as part of the service level agreement (SLA) by service provider. It is important for service provider to find a feasible set of component services to fulfill the response time requirement for composite service as promised. To address this problem, we propose a fully automated approach to synthesize the response time requirement for component services, given the response time requirement of composite service. Our approach is based on parameter synthesis techniques for real-time systems.

The proposed methods have been implemented in a series of software tools, to provide verification and analysis support for Web service composition.

**Key words: Web Service, Web Service Composition, Service Orchestration, Service Choreography, Model Checking, Partial Order Reduction, Formal Verification**

# List of Tables

2.1	Standards used by Web Services . . . . .	10
2.2	Semantics of LTL . . . . .	16
3.1	Syntax of Choreography . . . . .	23
3.2	Syntax of Orchestration . . . . .	27
3.3	WS@PAT vs WS-Engineer . . . . .	40
4.1	Performance evaluation on model checking <i>Orc's</i> model . . . . .	63
5.1	QoS Attribute Values . . . . .	73
5.2	Aggregation Function . . . . .	75
5.3	Experiment Results . . . . .	85



# List of Figures

1.1	Overall Picture . . . . .	5
3.1	A sample choreography . . . . .	24
3.2	Choreography structural operational semantics: where $\checkmark$ is the special event of termination . . . . .	25
3.3	A simple orchestration . . . . .	29
3.4	Choreography to orchestration projection function . . . . .	32
3.5	Definition of Initiating Roles . . . . .	33
3.6	Choreography repair function . . . . .	35
3.7	WS@PAT verification performance . . . . .	38
4.1	Partial Order Reduction . . . . .	44
4.2	Hierarchical Concurrent Processes . . . . .	44
4.3	Syntax of Orc . . . . .	46
4.4	HCP of a general hierarchical concurrent process . . . . .	51

4.5	HCP of General Orc Expressions . . . . .	53
4.6	An Orc Example . . . . .	53
4.7	Relation of Processes between $P$ and $P'$ . . . . .	53
4.8	Execution of <i>Orc</i> process $P = A   B$ . . . . .	54
4.9	LTS of <i>Orc</i> Process $P = (P_1   P_2), P_1 = ((1   2) \ll 3), P_2 = (4 \ll 6)$ . . . . .	56
5.1	Computer Purchasing Service . . . . .	71
5.2	LTS of CPS where $i_1$ is sInv(PBS), $i_2$ is sInv(CBS), $i_3$ is sInv(MS) and $i_4$ is sInv(SS) . . . . .	77
5.3	LTS of CPS with Availability and Cost, where $i_1$ is sInv(PBS), $i_2$ is sInv(CBS), $i_3$ is sInv(MS) and $i_4$ is sInv(SS) . . . . .	80
5.4	LTS of CPS with Response Time, Availability and Cost, where $i_1$ is sInv(PBS), $i_2$ is sInv(CBS), $i_3$ is sInv(MS) and $i_4$ is sInv(SS) . . . . .	81
6.1	Vehicle Booking Service . . . . .	92
6.2	Activation function . . . . .	100
6.3	Idling function . . . . .	101
6.4	Operational semantics . . . . .	102
6.5	LTS of service $\mathcal{M}$ . . . . .	104
6.6	LTS of composite service $S$ . . . . .	106
6.7	LTS of composite service $S'$ . . . . .	106
6.8	LTS of <i>VBS</i> . . . . .	109

# List of Algorithms

4.1	CAmple . . . . .	58
5.1	Algorithm <i>TagTime</i> ( $P, x$ ) . . . . .	82
5.2	Algorithm <i>CalculateTime</i> ( $P$ ) . . . . .	83
6.1	Algorithm <i>LocalTimeConstraint</i> ( $s_0$ ) . . . . .	108
6.2	Algorithm <i>synConsAOLTS</i> ( $s$ ) . . . . .	108

# Chapter 1

## Introduction

Service Oriented Architecture (SOA) represents an important design architecture nowadays. Web service technologies, as an SOA based on World Wide Web, have emerged as a de-facto standard for integrating disparate applications and systems using open, XML-based standards. Services perform functions ranging from answering simple requests to dealing with complex business processes. Services are self-describing, self-contained autonomous software system available via a network, such as the Internet. They are built in a way that is independent from the context, which means that service providers and service consumers are loosely coupled.

Web service composition makes use of existing service-based applications as components to achieve a business goal. The service that is composed by service composition is a *composite service* and services that the composite service makes use of are called *component services*. To guarantee the user satisfaction, there is often a contract, called service-level agreements (SLAs), which specifies the non-functional requirements that the service providers must obey. Testing approach can only mitigate this problem to a certain level. One of the prominent quotes from Dijkstra has reflected this fact: "Program testing can be used to



show the presence of bugs, but never to show their absence!" [38].

In business where services play a crucial role, a bug might cost millions dollars and service failures might cause loss of life. And service composition is inevitably rich in concurrency and it is not a simple task for programmers to utilize concurrency as they have to deal with multi-threads and critical regions. It is reported that among the common bug types concurrency bugs are the most difficult to fix correctly, the statistic shows that 39% of concurrency bugs are fixed incorrectly [95]. Since the complexity of service composition continues to escalate, an automated approach for verifying the functional and non-functional properties is desired.

## 1.1 Summary of this thesis

Although there have been a number of approaches for verifying and analyzing Web service composition. There are still some research gaps summarized as follows:

- Web service composition languages have been proposed in recent years, which can be categorized into two viewpoints – Web service choreography and Web service orchestration. Given a choreography and an orchestration of Web service composition that is not consistent to each other, there is no existing approach that could provide repair mechanism for repairing the orchestration to make it conform with the choreography.
- Service composition languages, such as Orc, possesses hierarchical concurrent structure. Existing verification of languages that have hierarchical concurrent structure does not take advantage of such structure for state-space reduction purpose.
- There is no existing work supports verification of combined functional and non-functional requirements of Web service composition, they only focus on verification

of one aspect, therefore, it cannot ensure two aspects of requirements at the same.

- Given the response time requirement of a composite service, there is no existing approach that could allow to synthesize the response time requirement of component services that are made use by the composite service.

In summary, existing works on verification and analysis of Web service composition are not complete and still have room to improve. Thus, the main goal of my research is to improve and refine the existing work to make it more complete and efficient. However, it is highly non-trivial to achieve this goal due to the reasons as follows:

- Choreography and orchestration are generally modeled in different languages/formalisms, and choreography models are even not executable, which increases the complexity to conformance checking.
- As the complexity and size of Web services continue to escalate, concurrency for Web service composition could lead to state-explosion, which poses a restriction on the sizes of the process to be analyzed.
- Given a Web service composition, there are many kinds of non-functional properties, eg., response time, availability, cost, different non-functional properties might have different aggregation functions for different compositional structures, and this poses a major challenge to integrate the non-functional properties into the functional verification framework.
- It is non-trivial to decompose the response time requirement of the composite service to component services since there are infinite number of ways for the decomposition to be done.

In this thesis, we address the above problems and challenges on verification and analysis of Web service composition. We summarize the contribution of this thesis as follows:

- We develop an algorithm based on refinement checking [79] to verify the conformance of Web service choreography and Web service orchestration. If these two views do not match, we further propose an algorithm to repair the Web service orchestration, such that after the repairing, Web service orchestration could conform to the Web service choreography. Abstract languages have been developed for this work to represent the service orchestration and choreography respectively, such that other orchestration or choreography languages could be translated to into abstract languages for conformance checking.
- We provide functional verification for the Web service composition language that is of the hierarchical concurrent nature. We propose a state-space reduction technique, called compositional partial order reduction (CPOR), to address the state explosion problem. CPOR has been shown to provide greater state-space reduction than classic partial order reduction methods in the context of hierarchical concurrent processes. Evaluation shows that CPOR is more effective in reducing the state space than classic partial order reduction methods. As a starting step, this work has been demonstrated and evaluated using *Orc* language [60]. The reason is that *Orc* language has simple and well-defined formal semantics, therefore the soundness could be easily shown.
- We provide integrated verification of functional and non-functional properties for Web service composition. To the best of our knowledge, we are the first work on such integration. We capture the semantics of Web service composition using labeled transition systems (LTSs) and verify the Web service composition directly without building intermediate or abstract models before applying verification approaches. We have evaluated this work using WS-BPEL language [56], which is the de-facto standard that is widely used for the description for Web service compositions.
- Given the response time of Web service composition, we develop a sound method to synthesize the local response time requirements for component services in the form

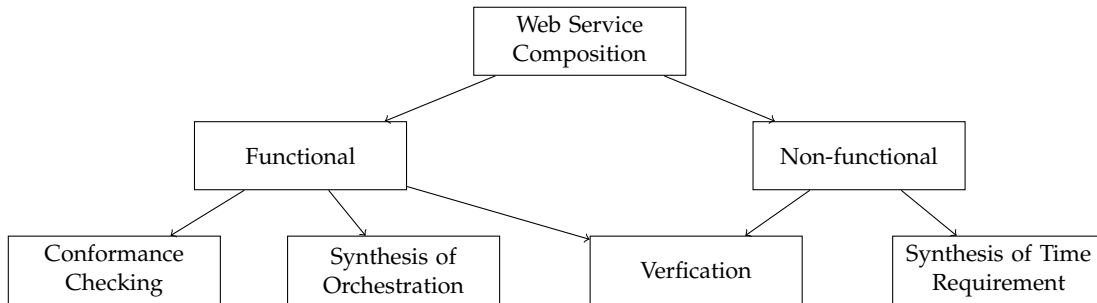


Figure 1.1: Overall Picture

of a set of constraints. The approach is implementation independent, therefore can be applied at the design stage of service composition. We have evaluated this work using WS-BPEL language.

The proposed approaches have been implemented in a series of software components such as WS module in PAT [84], and have been evaluated in several real-world case studies.

## 1.2 Overall Picture

Figure 1.1 describes the overall picture of this thesis. This thesis is focused on Web service composition. There are two important kinds of requirements of Web service composition, i.e., functional and non-functional requirements. For functional requirements, we check the conformance between orchestration and choreography and synthesize a prototype orchestration from the given choreography. For non-functional requirements, we synthesize the response time requirement for each component service by given the composite service's response time requirement. However, the two kinds of requirements are crucial to Web service composition, in order to guarantee both aspects, we check the combined functional and non-functional requirements.

## 1.3 Thesis Outline

In this section, we briefly present the outline of the thesis and the overview of each chapter.

Chapter 2 provides the background knowledge of this work. First, we introduce important features and concepts of Web service composition including service languages and functional and non-functional requirements. Second, model checking techniques are briefly introduced, and we also introduce properties specification which can be written in the form of linear temporal logic (LTL).

Chapter 3 presents model-based methods for automatic analysis of Web service compositions, in particular, linking two different views of Web services. We propose a method to mechanically synthesize a prototype Web service orchestration from choreography, by repairing the choreography if necessary and projecting relevant behaviors to each service provider.

Chapter 4 presents our approach in verifying a Web service composition language that is of hierarchical concurrent nature. We propose a new method, called Compositional Partial Order Reduction (CPOR), which aims to provide greater state-space reduction than classic partial order reduction methods in the context of hierarchical concurrent processes. Evaluation shows that CPOR is more effective in reducing the state space than classic partial order reduction methods.

Chapter 5 presents integrated functional and non-functional requirements verification of Web service composition, which makes use of the labeled transition systems (LTSs) directly from the semantics for functional verification. For non-functional properties, different strategies are used to integrate different non-functional properties into the functional verification framework.

Chapter 6 discusses a fully automatic approach to synthesize the response time require-

ment of component services, in the form of a constraint on the local response times, that guarantees the global response time requirement. Our approach is based on parameter synthesis techniques for real-time systems. It has been implemented and evaluated with real-world case studies.

Chapter 7 summarizes the thesis and discusses future research directions.

## 1.4 Acknowledgement of Published Work

Most of the work presented in this thesis has been published in international conference proceedings.

- **Model-based Methods for Linking Web Service Choreography and Orchestration** [85]. This paper was published at the 17th Asia-Pacific Software Engineering Conference (APSEC 2010). The work is presented in Chapter 3.
- **Verification of Computation Orchestration System with Compositional Partial Order Reduction** [90]. This paper was published at the 13th International Conference on Formal Engineering Methods (ICFEM 2011). The work is presented in Chapter 4.
- **Verification of Functional and Non-functional Requirements of Web Service Composition** [29]. This paper was published at the 15th International Conference on Formal Engineering Methods (ICFEM 2013). The work is presented in Chapter 5.
- **Dynamic Synthesis of Local Time Requirement for Service Composition** [88]. This paper was published at the 35th International Conference on Software Engineering (ICSE 2013). The work is presented in Chapter 6.

For all the publications mentioned above, I have contributed substantially in both theory development and tool implementation.



## Chapter 2

# Background

### 2.1 SOA and Web Service Composition

#### 2.1.1 SOA and Web Service

The reality in enterprise applications is that the infrastructure is heterogeneous across operating systems, application infrastructures, and system software. It is a challenging task to integrate the heterogeneous system to work as a whole. In addition, some old applications are tightly integrated with the existing business processes, and to build a new application from scratch is not a feasible option. Service Oriented Architecture (SOA) is proposed to address this problem.

Service Oriented Architecture (SOA) is a set of design principles for system development and integration. A *service* is a piece of application's business logic or individual functions that are modularized and presented to consumer applications. The major advantage of



Composition	WS-CDL, WE-BPEL, Orc
Description	WSDL
Message	SOAP
Transmission	HTTP, FTP, SMTP

Table 2.1: Standards used by Web Services

services is their loosely coupled nature — the interface is independent of the implementation. SOA with its loose coupling nature allows an enterprise to integrate their existing applications, and furthermore, to extend with new functionalities easily in response to business changes with agility. Web services technologies are a realization of SOA based on internet protocols such as HTTP. It is formally defined as *a software system designed to support interoperable machine-to-machine interaction over a network* [3].

The goal of Web service technology is to offer a communication bridge between the heterogeneous computational environments. This allows many business operations to be automated. Furthermore, since the communication is done through the World Wide Web, Web services could leverage on the ubiquitous internet connectivity for universal reach. To achieve this goal, a stack of protocols based on open and accepted standards (as shown in Table 2.1) are used. For example, at the transmission level Web services take advantage of HTTP, which is supported by most Web browsers and servers. Another enabling technology is XML (Extensible Markup Language) [4]. XML is a widely accepted standard for storing, carrying, and exchanging data. The core Web service standards comprise of SOAP, and WSDL, and both are specified in XML format. SOAP (Simple Object Access Protocol) [5] is a lightweight platform and language neutral communication protocol for Web services to communicate via standard internet protocols such as HTTP. WSDL (Web Services Description Language) [6] is used to define the interface of Web services, therefore the consumer applications know how to access them. Web services are a relatively new standards. To make it truly based on open and accepted standards, there are many aspects

of it (such as security) need to be standardized. Therefore, there are a number of WS-\* specifications [1] (e.g. WS-BPEL, WS-Addressing, WS-Security, WS-Resource and so on) dealing for other aspects of Web service usage: composition, addressing, security, resource states, and so on. We will focus on Web service composition in this paper, and it is discussed in the next section 2.1.2.

### 2.1.2 Web Service Composition

While the technology for creating services and interconnecting them with a point-to-point basis has achieved a certain degree of maturity, it remains a challenge to integrate multiple services for complex interactions. Service composition makes use of existing services based applications as components to achieve a business goal. The service that is composed by service composition is called a *composite* service, and services that the composite service makes use of are called *component* services.

#### 2.1.2.1 Service Orchestration and Service Choreography

Web service composition standards are proposed in order to address this challenge. Web service composition could be categorized into two categories, which are Web service orchestration and Web service choreography. Their differences are mainly in their viewpoints of the composition. Web service *orchestration* refers to Web service descriptions which take a *local* point of view. That is, an orchestration describes collaborations of the Web services in predefined patterns based on the local decision about their interactions with one another at the execution level.

A representative is WS-BPEL (short for Web Service Business Process Execution Language [56]), which models business processes by specifying the workflows of carrying

out business transactions. It provides basic activities such as service invocation, and compositional activities such as sequential and parallel composition to describe the composition of Web services.

Another example is Orc [60], which is designed to specify orchestrations and wide-area computations in a concise and structured manner. It has four concurrency combinators, which can be used to manage timeouts, priorities, and failures effectively. The standard operational semantics [93] of Orc support highly concurrent executions of Orc sub-expressions.

Web service *choreography* is referred to Web service specification which describes collaboration protocols of cooperating Web service participants from a *global* point of view. An example is WS-CDL (short for Web Service Choreography Description Language [27]).

#### 2.1.2.2 Functional and Non-Functional Requirement

There are two kinds of requirements of Web service composition, i.e., functional and non-functional requirements. Functional requirements focus on the functionalities of the Web service composition, which detail the operational characteristics that define the overall behavior of the service. Given a booking service, an example of functional requirements is that a flight ticket with price higher than \$2000 will never be purchased. The non-functional requirements are concerned with the Quality of Service (QoS).

QoS refers to the ability of the Web service to respond to expected invocations and to perform them at the level commensurate the mutual expectation of both services' providers and consumers. QoS has become an important criterion which determines the usability and of utility of service.

Non-functional requirements are often recorded in service-level agreements (SLAs), which is the contractual basis between service consumers and service providers on the expected

quality of service (QoS) level. Given a booking service, an example of non-functional requirements is that the service will respond to the user within 4 seconds. Typical non-functional requirements include response time, availability, cost and so on.

## 2.2 Basics of Model Checking

Nowadays, testing and debugging have been widely used in programmers. However, there are often some bugs that are difficult to find, not mention to fix. It is limited for the ability of most diligent and faithful testing techniques to explore all scenarios and to find all possible bugs. Testing techniques only can help to find bugs, but not to show the correctness of the program. One of most prominent quotes from Dijkstra mentioned that "Program testing can be used to show the presence of bugs, but never to show their absence!". Software verification [18] is one of the techniques for checking the correctness of software (i.e., source code), because it automatically traverses all scenarios of the target system.

Model checking [34] is one approach of software verification. It is an automatic technique for verifying finite state concurrent systems. Since it is a verification technique that exhaustively explores all possible system states, it is feasible for systems with finite states. The performance of model checking approach is related to the size of the system's state space. The process of model checking consists of several tasks. First of all, the system design is required to be converted into a formalism that can be accepted by model checking tools. Requirements of the systems are abstracted as logic specifications so that it can be better understood and verified by model checking tools. One common example of logic specifications is temporal logic, which can assert how the behavior of the system evolves over time. If the verification result is negative, users are often provided with a witness trace (or counterexample). The analysis of the error trace may require modifications to the

model and repeat the model checking process. Each process of the model checking, namely modeling, specification and verification, will be explained in the following sections.

## 2.3 System Modeling

The first step of the model checking is to convert the system design into a formalism that can be accepted by model checking tools, it is crucial and key to the model checking. However, sometimes it is not simple due to the limitation of memory and time. We might need the higher level abstraction and at the same time the important issues should be kept, unnecessary details are required to be eliminated. Therefore, it is not a simple task to model the system. For Web services, we may focus on the communication between services, while ignore the actual contents. In different models, system states may be different in order to capture the special features of the model.

A system *state* is a snapshot of the system to capture features of the system at a particular instant of time, sometimes, we also call it *configuration*. Changes in system state may be triggered by some action, which we call state transition. A state transition is described by giving the state before the action, the state after the action and the action. A computation of the system can be defined by a sequence of finite or infinite states. We use a state transition graph called a *Kripke structure* [28] to model a system formally.

**Definition 2.3.1** (Kripke structure). *Let  $AP$  be a non-empty set of atomic propositions. A Kripke structure  $M$  over a set of atomic propositions  $AP$  is a four-tuple  $M = (S, S_0, R, L)$ , where*

- $S$  is a finite set of states;
- $S_0 \subseteq S$  is the set of initial states;
- $R \subseteq S \times S$  is a transition relation;

- $L : S \rightarrow 2^{AP}$  is a function that labels each state with the set of atomic propositions hold in this state.

## 2.4 Specification and Verification

Specification is the properties that the system must satisfy. There are many different ways to describe properties. In the state-based model, one common way to express properties is temporal logics which can capture the behavior of the system evolve over time. In our work, we support the linear temporal logic (LTL) formulas. There are two kinds of properties: safety properties and liveness properties. In this section, we will introduce these two properties and the algorithms for checking them.

### 2.4.1 Safety Property

A safety property is a property stating that "something bad does never happen". In other words, safety properties are used for verifying whether undesirable behaviors will happen or not. For example, "The temperature of reactor will never exceed 100°C". Intuitively, a property  $\varphi$  is a safety property if each violation of  $\varphi$  occurs after a finite execution of the system. In general, safety specifications include the absence of deadlocks and unreachability of states that are not supposed to be reached. *Deadlock* of a state means that there is no outgoing transitions for the state in a model, which means that terminal state has been reached. For example, in a mutual exclusion algorithm, there are two processes and both are waiting for the resource, then the deadlock occurs. Deadlock is common problem in multiprocessing systems and concurrent systems, where shared resources are usually restricted to only one process in order to avoid conflict.

To verify safety properties, it is usually to conduct a depth first search (or breadth first

Operator	Name	Explanation
$\varphi_1 \wedge \varphi_2$	Union	Satisfy $\varphi_1$ and $\varphi_2$ at the same time.
$X\varphi$	Next	Next state satisfies $\varphi$ .
$\Box\varphi$	Globally	Always satisfy $\varphi$ .
$\Diamond\varphi$	Finally	Eventually reach a state satisfy $\varphi$ .
$\varphi_1 \cup \varphi_2$	Until	$\varphi$ should hold the subsequent path at least until $\varphi_2$ starts to hold.

Table 2.2: Semantics of LTL

search) in the state space. During the search, if the reached state is undesirable, a counterexample will be given.

### 2.4.2 Liveness Property

Different from the safety property, a liveness property is a property stating that "something good eventually happens", and liveness properties are violated in infinite time by infinite runs, while safety properties are in finite time. In a mutual exclusion algorithm, a liveness property example is that "each process will eventually access the critical section".

Temporal Logics could be used to express liveness properties. Examples of temporal logics include Computation Tree Logic (CTL) [31], Linear Temporal Logic (LTL) [82] and CTL\* [32]. In this thesis, we use LTL to model liveness properties.

**Definition 2.4.1** (Syntax of LTL). *Let  $P$  be a set of atomic propositions. A LTL formula is:*

$\varphi ::= true \mid false \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 \cup \varphi_2$ , where  $a \in P$ .

As described in Table 2.2,  $\varphi_1 \wedge \varphi_2$  means properties  $\varphi_1$  and  $\varphi_2$  are satisfied at the same time.  $X\varphi$  means that  $\varphi$  holds on the next state.  $\Box\varphi$  means that  $\varphi$  holds on at the all state in the execution.  $\Diamond\varphi$  means that  $\varphi$  is satisfies in some state of the execution.  $\varphi_1 \cup \varphi_2$  means that that  $\varphi_1$  has to hold at least until  $\varphi_2$  starts to hold.

By combining the operators, new temporal operators are obtained. For example,  $\Box\Diamond\varphi$  describes "always eventually  $\varphi$ ", which stating that at any moment  $j$  there is a moment  $i \geq j$  where  $\varphi$  is satisfied.  $\Diamond\Box\varphi$  describes "eventually always  $\varphi$ ", which stating that from a moment  $j$ ,  $\varphi$  is always satisfied in the later execution.

Explicit state model checking converts the system into a automaton  $\mathcal{M}$ , which represents a Kripke structure with nodes for states and edges for transitions. In addition, the negation of LTL specification  $\varphi$  is also translated into a corresponding Büchi automaton  $\mathcal{A}_{\neg\varphi}$ . Subsequently, the emptiness of the product of  $\mathcal{M}$  and  $\mathcal{A}_{\neg\varphi}$  is checked. If the product is not empty, then system  $\mathcal{M}$  does not satisfy property  $\varphi$ , a counterexample will be reported. Otherwise, property  $\varphi$  holds in the system.

*On-the-fly* model checking [35, 54] is adopted in this thesis to reduce the state space to be explored. Instead of constructing the automaton for  $\mathcal{M}$  and  $\mathcal{A}_{\neg\varphi}$  separately, we construct the property automaton  $\mathcal{A}_{\neg\varphi}$  first, and use that to guide the construction of system automaton  $\mathcal{M}$ . An advantage of the on-the-fly approach to model checking is that, some states in system automaton  $\mathcal{M}$  may never be generated at all. In addition, once a counterexample is found, there is no need to complete the construction of the product, which could vastly reduce the time for finding a counterexample.





## Chapter 3

# Conformance Checking of Service Composition

The Web services paradigm promises to enable rich, dynamic, and flexible interoperability of highly heterogeneous and distributed Web-based platforms. In recent years, many Web service composition languages have been proposed. There are two different viewpoints, and correspondingly two terms, in the area of Web service composition. Web service *choreography* is usually referred to Web service specification which describes collaboration protocols of cooperating Web service participants from a global point of view. An example is WS-CDL (short for Web Service Choreography Description Language [27]). Web service *orchestration* refers to Web service descriptions which take a local point of view. That is, an orchestration describes collaborations of the Web services in predefined patterns based on local decision about their interactions with one another at the message/execution level. A representative is WS-BPEL (short for Web Service Business Process Execution Language [56]), which models business processes by specifying the work flows of carrying out business transactions.

Informally, a choreography may be viewed as a contract among multiple corporations, i.e., a specification of requirements (which may not be executable). An orchestration is the composition of concrete services provided by each corporation who realizes the contract. The distinction between choreography and orchestration resembles the well studied distinction between sequence diagrams (which describes inter-object system interactions, taking a global view) and state machines (which may be used to describe intra-object state transitions, taking a local view). Likewise, there are two important problems to be addressed.

One is the *verification* problem, i.e., to verify whether a choreography or an orchestration is correct with respect to critical system properties or whether they are consistent with each other. The latter means that the orchestration faithfully implements all and only what the contract states. The other one is the *synthesis* problem, i.e., to decide whether a choreography can be realized by any orchestration (referred as implementable) and synthesize a prototype orchestration if possible.

The solutions to both problems are important in the development of Web services. Solving either problem is however highly non-trivial. Firstly and most importantly, choreography and orchestration are generally modeled in different languages/formalisms, and choreography models are even not executable. Hence, there is natural gap between the two views. To perform effective analysis on the two views, we need to bridge the gap. Secondly, ideally it is sufficient to verify a single Web service invocation which is independent of other service invocations. In reality, this is often not true because of physical constraints (see [43], like the number of Web service instances are bounded by the thread pool size of the underlying operating system). As a result, multiple service invocations must be verified as a whole. Because Web services are designed for potentially large number of users (who may invoke the services simultaneously), verifying Web services based on model checking techniques must cope with state space explosion due to concurrent service invocations. Lastly, synthe-

sizing orchestration from choreography resembles the distributed synthesis problem (e.g., in the setting of sequence diagrams), which has been shown to be undecidable in general and in many restrictive settings [75]. Worse, synthesizing a distributed object system with the exact behaviors is impossible if there are implied scenarios [13]. Both results apply to Web service choreography (see examples later).

In this work, we offer practical solutions to both problems using a model based approach. First of all, we propose formal languages for modeling choreography and orchestration respectively with formal operational semantics. This creates a unified semantics model for the two views, which allows communications between choreography and orchestration models. To make them practical, these languages cover many language constructs for Web service compositions (i.e., behavioral aspects of WS-CDL and WS-BPEL).

In order to verify Web services under physical constraints, on-the-fly model checking techniques are adopted and extended specially to handle multiple concurrent service invocations. Consistency between choreography and orchestration is verified by showing conformance relationship (i.e., trace inclusion) between the choreography and the orchestration. Based on the refinement checking [79], we develop a verification algorithm to support data communications between choreography and orchestration, which allows orchestration to drive the execution of (non-executable) choreography. It is further optimized for Web services.

In order to deal with undecidability of the synthesis problem, we adopt a scalable lightweight approach. We do not claim to solve the problem completely, instead, we present a practical way to avoid undecidability. That is, instead of semantically checking whether a choreography is distributively implementable or not, we apply static analysis (based on the syntax) to check whether the choreography satisfies certain sufficient condition for being implementable. If positive, a synthesis procedure is invoked to automatically generate an orchestration prototype. Otherwise, we go further by using a repairing process to generate

an implementable choreography by inserting communications between service providers. The repaired choreography may provide hints on how to correct the original one. Lastly, our engineering efforts have realized the methods in a toolkit named WS@PAT (available at <http://pat.comp.nus.edu.sg>), which is a self-contained framework for Web service modeling, simulation, verification and synthesis.

**Chapter Outline.** Section 3.1 presents the modeling language. Given a choreography and an orchestration, Section 3.2 shows an approach of checking their consistency. Given the orchestration is not consistent with the choreography, Section 6 introduces our methodology in synthesizing a new orchestration that is consistent with the choreography, provided that the choreography satisfies certain conditions. Section 3.4 demonstrates the implementation of our methods. Section 3.5 surveys the related work.

## 3.1 Modeling

In this section, we present modeling languages which are expressive enough to capture all core features of Web service choreography and orchestration. There are two reasons for introducing intermediate modeling languages for Web services. First, heavy languages like WS-CDL or WS-BPEL are designed for machine consumption and therefore are lengthy and complicated in structure. Moreover, there are mismatches between WS-CDL and WS-BPEL. For instance, WS-CDL allows channel passing whereas WS-BPEL does not. The intermediate languages focus on the interactive behavioral aspect. The languages are developed based on previous works of formal models for WS-CDL and WS-BPEL [27, 78, 76]. Second, based on the intermediate languages and their semantic models (namely, labeled transition systems), our verification and synthesis approaches is not bound to one particular Web service language. For instance, newly proposed orchestration languages like Orc [60] is also supported in our tool. This is important because Web service languages

evolve rapidly. Being based on intermediate languages allows us to quickly cope with new syntaxes or features (e.g., by tuning the preprocessing component).

### 3.1.1 Choreography: Syntax and Semantics

The following is the core syntax for modeling interactive behaviors of Web service choreography, e.g., in WS-CDL.

$\mathcal{I}$	::=	$Stop \mid Skip$	inaction and termination
		$svr(A, B, \tilde{c}h) \rightarrow \mathcal{I}$	service invocation
		$svr(A, B, exp) \rightarrow \mathcal{I}$	channel communication
		$x := exp; \mathcal{I}$	assignment
		$if\ b\ \mathcal{I}\ else\ \mathcal{J}$	conditional
		$\mathcal{I} \square \mathcal{J}$	choice
		$\mathcal{I} \parallel \mathcal{J}$	service interleaving
		$\mathcal{I}; \mathcal{J}$	sequential

Table 3.1: Syntax of Choreography

In WS@PAT, we support user-defined data types and dynamic invocation of C# library and hence modeling data components of Web services are feasible. For simplicity, we skip details on data variables in this paper. Let  $\mathcal{I}$  (short of *interaction*),  $\mathcal{J}$  be terms of choreography. Let  $A, B$  range over Web service roles;  $ch$  range over communication channels;  $svr$  range over a set of pre-setup service invocation channels (refer to discussion later);  $\tilde{c}h$  denote a sequence of channels;  $x$  range over variables;  $exp$  be an expression and  $b$  be a predicate over only the variables.

We assume that each role is associated with a set of local variables and there are no globally shared variables among roles. This is a reasonable assumption as each role (which is a service) may be realized in a remote computing device. Informally,  $svr(A, B, \tilde{c}h)$ , where  $svr$  is pre-defined service invocation channel, states that role  $A$  invokes a service provided by role  $B$  through channel  $svr$ . A service invocation channel is the one that is registered with

1.  $BuySell() = B2S(Buyer, Seller, \{Bch\}) \rightarrow Session();$
2.  $Session() = Bch(Buyer, Seller, QuoteRequest) \rightarrow Bch(Seller, Buyer, QuoteResponse.x) \rightarrow$
3.  $if (x \leq 1000) \{$
4.      $Bch(Buyer, Seller, QuoteAccept) \rightarrow Bch(Seller, Buyer, OrderConfirmation) \rightarrow$
5.      $S2H(Seller, Shipper, \{Bch, Sch\}) \rightarrow$
6.      $(Sch(Shipper, Seller, DeliveryDetails.y) \rightarrow Stop \parallel Bch(Shipper, Buyer, DeliveryDetails.y) \rightarrow Stop)$
7.  $\} else \{ Bch(Buyer, Seller, QuoteReject) \rightarrow Session() \square Bch(Buyer, Seller, Terminate) \rightarrow Stop \};$

Figure 3.1: A sample choreography

a service repository so that the service is subject for invocation.  $\tilde{ch}$  is a sequence of session channels which are created for this service invocation only. Notice that because the same service shall be available all the time, service channel  $svr$  is reserved for service invocation only.  $ch(A, B, exp)$  where  $ch$  is a session channel states that role  $A$  sends the message  $exp$  to role  $B$  through channel  $ch$ .

$x := exp$  assigns the value of  $exp$  to variable  $x$ . Without loss of generality, we always require that the variables constituting  $exp$  and  $x$  must be associated with the same role. If  $b$  evaluates to true,  $if b I else J$  behaves as  $I$ , otherwise  $J$ . Given a variable  $x$  (a condition  $b$ ), we write  $role(x)$  ( $role(b)$ ) to denote the associated role.  $I \square J$  is an unconditional choice (i.e., choice of two unguarded working units in WS-CDL) between  $I$  and  $J$ , depending on whichever executes first.  $I \parallel J$  denotes two interactions running in parallel. Notice that there are no message communications between  $I$  and  $J$ . Two choreographies executing in a sequential order is written as  $I; J$ . We remark that recursion is supported by referencing a choreography name.

The syntax above is expressive enough to capture the core Web service choreography features. For instance, channel passing is supported as we are allowed to transfer a sequence of channels on service invocation. Fig. 3.1 presents a choreography of an online store. The choreography coordinates three roles (i.e., *Buyer*, *Seller* and *Shipper*) to complete a business transaction among two pre-defined services channel  $B2S$  and  $S2H$ . At line 1, the *Buyer* communicates with the *Seller* through service channel  $B2S$  to invoke its service.

$$\begin{array}{c}
\frac{}{(svr(A, B, \tilde{ch}) \rightarrow \mathcal{I}, V) \xrightarrow{svr! \tilde{ch}} (svr?(B, \tilde{ch}) \rightarrow \mathcal{I} \parallel svr(A, B, \tilde{ch}) \rightarrow \mathcal{I}, V)} [inv1] \\
\frac{}{(svr?(B, \tilde{ch}) \rightarrow \mathcal{I}, V) \xrightarrow{svr? \tilde{ch}} (\mathcal{I}, V)} [inv2] \qquad \frac{}{(ch(A, B, exp) \rightarrow \mathcal{I}, V) \xrightarrow{ch! v} (ch?(B, v) \rightarrow \mathcal{I}, V)} [ch1] \\
\frac{}{(ch?(B, v) \rightarrow \mathcal{I}, V) \xrightarrow{ch?v} (\mathcal{I}, V)} [ch2] \qquad \frac{eval(exp, V) = v}{(x := exp; \mathcal{I}, V) \xrightarrow{\tau} (\mathcal{I}, V' \oplus x \mapsto v)} [assign] \\
\frac{(\mathcal{I}, V) \xrightarrow{e} (\mathcal{I}', V'), eval(b, V) = true}{(if b \mathcal{I} else \mathcal{J}, V) \xrightarrow{e} (\mathcal{I}, V')} [b1] \qquad \frac{(\mathcal{J}, V) \xrightarrow{e} (\mathcal{J}', V'), eval(b, V) = false}{(if b \mathcal{I} else \mathcal{J}, V) \xrightarrow{e} (\mathcal{J}, V')} [b2] \\
\frac{(\mathcal{I}, V) \xrightarrow{e} (\mathcal{I}', V')}{(\mathcal{I} \square \mathcal{J}, V) \xrightarrow{e} (\mathcal{I}', V')} [choice1] \qquad \frac{(\mathcal{J}, V) \xrightarrow{e} (\mathcal{J}', V')}{(\mathcal{I} \square \mathcal{J}, V) \xrightarrow{e} (\mathcal{J}', V')} [choice2] \qquad \frac{(\mathcal{I}, V) \xrightarrow{e} (\mathcal{I}', V')}{(\mathcal{I} \parallel \mathcal{J}, V) \xrightarrow{e} (\mathcal{I}' \parallel \mathcal{J}, V')} [inter1] \\
\frac{(\mathcal{I}, V) \xrightarrow{e} (\mathcal{I}', V')}{(\mathcal{I} \parallel \mathcal{J}, V) \xrightarrow{e} (\mathcal{I}' \parallel \mathcal{J}, V')} [inter2] \qquad \frac{(\mathcal{I}, V) \xrightarrow{e} (\mathcal{I}', V'), e \neq \checkmark}{(\mathcal{I}; \mathcal{J}, V) \xrightarrow{e} (\mathcal{I}'; \mathcal{J}, V')} [seq1] \qquad \frac{(\mathcal{J}, V) \xrightarrow{\checkmark} (\mathcal{J}', V')}{(\mathcal{I}; \mathcal{J}, V) \xrightarrow{\tau} (\mathcal{J}, V')} [seq2]
\end{array}$$

Figure 3.2: Choreography structural operational semantics: where  $\checkmark$  is the special event of termination

Channel  $Bch$  which is sent along the service invocation is to be used as a session channel for the session only. In the *Session*, the *Buyer* firstly sends a message *QuoteRequest* to the *Seller* through channel  $Bch$ . At line 2, the *Seller* responds with some quotation value  $x$ , which is a variable. Notice that in choreography, the value of  $x$  may be left unspecified at this point. At line 5, the *Seller* sends a message through the service channel  $S2H$  to invoke a shipping service. Notice that the channel  $Bch$  is passed onto the *Shipper* so that the shipper may contact the *Buyer* directly. At line 6, the *Shipper* sends delivery details to the *Buyer* and *Seller* through the respective channels. The rest is self-explanatory.

In this work, we focus on the operational semantics. Given a choreography model, a system configuration is a 2-tuple  $(\mathcal{I}, V)$ , where  $\mathcal{I}$  is a choreography and  $V$  is a mapping from the variables to their values, i.e., from data variables to their valuations or from channel variables to channel instances. A transition is expressed in the form of  $(\mathcal{I}, V) \xrightarrow{e} (\mathcal{I}', V')$ . The transition rules are presented in Fig. 3.2. Rule *inv1* captures service invocation, where event  $svr! \tilde{ch}$  occurs. Afterwards, rule *inv2* becomes applicable so that the service invoking



request is ready to be received. At the same time, a copy of the choreography is forked. This is because a service may be invoked multiple times, possibly simultaneously, by different service users and all service invocations must conform to the choreography. In fact, in the standard practice of Web services, a service is embodied by a shared channel in the form of URLs or URIs through which many users can throw their requests at any time. For instance, different processes acting as *Buyers* may invoke the service provided by the *Seller*. All *Buyers* must follow the communication sequence. Furthermore, in order to match the reality, we assume that both service invocation and channel communication are asynchronous in this work. As a result, service invocation (or channel communication) is divided into two events, i.e, the event of issuing a service invocation (or channel output) and the event of receiving a service invocation (or channel input). This is captured by rules *inv1*, *inv2*, *ch1* and *ch2*. For simplicity, we assume that a function *eval* returns the value of an expression *exp* given the valuation of variables *V*. Rule *assign* updates variable valuations. The rest of the rules resembles those for the classic CSP [52]. Notice that an assignment results in a invisible transition (written as  $\tau$ ). Only communication are visible.

Given a choreography  $\mathcal{I}$ , we build a Labeled Transition System (LTS)  $(S, init, T)$  where  $S$  is the set of reachable configurations, *init* is the initial state (i.e., the initial choreography and the initial valuation of the variables) and  $T$  is a labeled transition relation defined by the semantics rules. A run of the LTS is a finite sequence of alternating configurations/events  $\langle s_0, e_0, s_1, e_1, \dots, e_{n-1}, s_n \rangle$  such that  $s_0$  is *init* and  $(s_i, e_i, s_{i+1}) \in T$  for all  $i : 0..n$ . A trace of  $\mathcal{I}$  is a finite sequence of events  $\langle e_0, e_1, \dots, e_k \rangle$  if and only if there is a run of the LTS  $\langle s_0, x_0, s_1, x_1, \dots, x_{n-1}, s_n \rangle$  such that  $\langle x_0, \dots, x_{n-1} \rangle \upharpoonright \{\tau\} = \langle e_0, \dots, e_k \rangle$  where  $\upharpoonright$  is the filtering operation to remove all  $\tau$  transitions (i.e., invisible events). The set of all finite traces of  $\mathcal{I}$  is denoted as  $traces(\mathcal{I})$ .

In order to verify properties about the choreography, we use model checking techniques to explore all traces of the transition system. One complication is that the choreography's

behavior may depend on environmental input which is only known during runtime with the execution of an orchestration. For instance, the price quote provided by the *Seller* is unknown given only the choreography in Fig. 3.1. We discuss this issue in Section 3.2.

### 3.1.2 Orchestration: Syntax and Semantics

$P ::=$	$Stop \mid Skip$	primitives
	$  \text{inv!}\tilde{ch} \rightarrow P$	service invoking
	$  \text{inv?}\tilde{x} \rightarrow P$	service being invoked
	$  \text{ch!exp} \rightarrow P$	channel output
	$  \text{ch?x} \rightarrow P$	channel input
	$  x := \text{exp}; P$	conditional branching
	$  \text{if } b \text{ } P \text{ else } Q$	service interleaving
	$  P \square Q$	orchestration choice
	$  P \Delta Q$	interrupt
	$  P \parallel Q$	interleaving
	$  P; Q$	sequential

Table 3.2: Syntax of Orchestration

A Web service orchestration  $\mathcal{O}$  is composed of multiple roles, each of which is specified as an individual process defined using the syntax above. A slightly different syntax is used to build orchestration models. The reason is that orchestration takes a local view and therefore all primitive actions are associated with a single role. Let  $P$  and  $Q$  be the processes, which describe behaviors of a role.

Process  $\text{inv!}\tilde{ch} \rightarrow P$  invokes a service (e.g., <invoke> in BPEL) through service channel  $\text{inv}$  and then behaves as specified by  $P$ . Or a service can be invoked by  $\text{inv?}\tilde{x} \rightarrow P$  where  $\tilde{x}$  is a sequence of channel variables which store the received channels. A process may send (receive) a message through a channel  $\text{ch}$  by  $\text{ch!exp} \rightarrow P$  ( $\text{ch?x} \rightarrow P$ ). Further, choice  $\square$  and interrupt  $\Delta$  can be used to model event/exception handler in languages like BPEL, e.g.  $\text{LoginProgram} \Delta \text{LoginExceptionHandler}$ . To match the reality, we always assume that the communication channels between different processes are asynchronous (and with a fixed

buffer size) in this work. The rest are similar to those of choreography.

Similarly, we define the operational semantics. Let  $V_A$  be the valuation of the variables associated with the role  $A$ . Let  $C$  be a valuation function of the channels, which maps a channel to the sequence of items in the buffer.  $C$  is a set of tuples of the form  $c \mapsto m\tilde{s}g$ . A configuration of the process is a 3-tuple  $(P, V_A, C)$ . The firing rules are skipped for the sake of space. We remark that as in choreography, service invocation in orchestration forks a new copy of the service and thus allows potentially many concurrent service invocations. In reality, however, the number of overlapping service invocations is bounded by the maximum number of threads the underlying operating system allows [43]. In next section, we discuss how to capture this constraint and at the same time perform efficient verification.

Because an orchestration is the cooperation of multiple roles or processes, behaviors of the processes must be composed in order to obtain the global behavior. Assume that  $P$  plays the role  $A$  in the orchestration is written as  $P@A$ . Given two processes, e.g.,  $P$  and  $Q$ , playing different roles, e.g.,  $A$  and  $B$ , the composition is  $P@A \parallel Q@B$ . The semantic rules for process composition is straightforward, i.e., a global step is constituted of a local step by either  $P$  or  $Q$ . Following the rules, given an orchestration with multiple roles, each of which is specified as a process defined above, we may build a LTS. The executions of the orchestration equal to the executions the LTS. Similarly, we define traces of an orchestration as  $\tau$ -filtered traces of the LTS. Given an orchestration  $O$ , let  $traces(O)$  be the set of finite executions.

Fig. 3.3 presents an orchestration which implements the choreography in Fig. 3.1. Each role is implemented as a separate component. Each component contains variable declarations (optional) and process definitions. We assume that the process *Main* defines the computational logic of the role after initialization. We remark that the orchestration generally contains more details than the choreography, e.g., the variable *counter* in *Buyer* constraints the number of attempts the buyer would try before giving up.

```

Role Buyer {var counter = 0;
  Main()      = B2S!{bch} → Session();
  Session()   = bch!QuoteRequest → counter++; bch?QuoteResonse.x →
               if (x <= 1000){ bch!QuoteAccept → bch?OrderConfirmation → bch?DeliveryDetails.y → Stop }
               elseif (counter > 3) {bch!QuoteReject → Session()} else {Stop}; }
Role Seller {var x = 1200;
  Main()      = B2S?{ch} → Session();
  Session()   = ch?QuoteRequest → ch!QuoteResonse.x → (ch?QuoteAccept → ch!OrderConfirmation →
               S2H!{ch, Sch} → Sch?DeliveryDetails.y → Stop □ ch?QuoteReject → Session()); }
Role Shipper {var detail = "20/10/2009";
  Main()      = S2H?{ch1, ch2} → (ch1!DelieryDetails.detail → Stop ||| ch2!DelieryDetails.detail → Stop); }

```

Figure 3.3: A simple orchestration

## 3.2 Verification

An orchestration can be verified against critical system properties like temporal properties or a choreography. We remark that service verification is performed under the physical constraints (e.g., a service may be blocked after the thread pool is full) in this work. In WS@PAT, we support full LTL formulae composed of propositions on data variables or events (e.g., a channel input/output, a local action, etc.). We adapt the automata-based on-the-fly approach to verify LTL formulae, i.e., by firstly translating a formula to a Büchi automaton and then check emptiness of the product of the system and the automaton. The details can be found in [84].

In the following, we define conformance between a choreography and an orchestration based on trace refinement and present an approach to verify it by showing refinement relationships. An orchestration  $O$  is valid w.r.t. a choreography  $I$  if and only if  $O$  refines  $I$  i.e.,  $traces(O) \subseteq traces(I)$ . As discussed above, both choreography and orchestration can be translated into LTSs. By the assumption that both the ranges of the variables and sizes of channels are finite and the number of concurrent service invocations are bounded, the LTSs have finite number of states. As a result, we can extend the refinement checking algorithm proposed in [79] to do the conformance checking.

A main challenge for verifying practical Web services by model checking is state space explosion. There are multiple causes of state space explosion. Two of them are 1) the numerous different interleaving of processes executing concurrently in service orchestration and 2) the large number of concurrent service invocations. In the following, we discuss two optimization techniques which have been adopted to cope with the above issues.

Firstly, the algorithm is improved with partial order reduction, to reduce the number of possible interleaving (particularly for orchestration). Events performed by single service role (e.g., local variable updates in service choreography or orchestration) are often independent with the rest of the system and hence are subject to reduction. During model checking, if a local action which results in a  $\tau$ -transition is enabled (together with actions performed by other roles), we only expand the system graph using this action and postpone the rest. By this way, we build a smaller LTS and therefore checks deadlock-freeness or LTL more efficiently. For refinement checking, we apply this reduction in two ways. One is to apply partial order reduction separately to invisible events of either the choreography or the orchestration. Notice that this reduction is trace preserving and therefore is sound for refinement checking.

Secondly, by a simple argument, it can be shown that  $\parallel$  is symmetric and associative. Naturally, different invocations of the same Web service are similar or even identical. By the above laws, the interleaving of multiple choreographes can be sorted (in certain fixed ordering) without changing the system behaviors. Therefore, if the choreography is in the form of  $\mathcal{I} \parallel \dots \parallel \mathcal{I} \parallel \dots$ , it is equivalent whether the first  $\mathcal{I}$  makes a transition or the second does. For verification of deadlock-freeness, safety or liveness properties, it is thus sound to pick one of the transitions and ignore the others. In general, this reduction could reduce the number of states up to the factor of  $N!$  where  $N$  is the number of identical components. This reduction is inspired by research on model checking parameterized systems [55] and [40].

There are a number of other algebraic laws which may help to reduce the number of states

(e.g.,  $\mathcal{I} \sqcap \mathcal{J} = \mathcal{J} \sqcap \mathcal{I}$ ). Nonetheless, it is a balance between the computational overhead (for the additional checking) and gain in state reduction. In our implementation (refer to Section 3.4), a set of specially chosen algebraic laws are used to detect equivalence of system configurations.

### 3.3 Prototype Synthesis

Given a choreography as a contract among multiple organizations, it is vital to guarantee that not only the contract is implementable but also it can be implemented in a non-ambiguous way. The synthesis problem of the classic sequence diagrams has been studied extensively [13, 25]. The negative results apply to the synthesis problem of Web service choreography. For instance, the following demonstrates the problem of implied scenarios in the setting of choreography. Assume that  $ch$  is an asynchronous session channel (with buffer size more than 2),  $A$  and  $B$  are two participating roles and  $M_1, M_2$  are two messages.

$$\begin{aligned} \mathcal{I}_{\text{exa}} &= (ch(A, B, M_1) \rightarrow ch(B, A, M_2) \rightarrow \text{Stop}) \\ &\quad \square (ch(B, A, M_1) \rightarrow ch(A, B, M_2) \rightarrow \text{Stop}) \end{aligned}$$

The specification states that either  $A$  sends  $M_1$  to  $B$  first and then  $B$  responds by  $M_2$ , or the system works the other way around. Exactly as in the setting of sequence diagram [13], the above choreography  $\mathcal{I}$  is not implementable because any distributed implementation would allow the following trace,

$$\langle ch!M_1, ch!M_2, ch?M_2, ch?M_1 \rangle$$

where  $ch!M_1$  is the event of ( $A$ ) sending message  $M_1$ .

Exactly telling whether a choreography is implementable or synthesizing a minimally restrictive prototype is expensive. Therefore, we follow and extend the work presented

$$\begin{array}{lcl}
\text{Stop} \triangleright X = \text{Skip} \triangleright X & = & \text{Stop} \\
(\text{svr}(A, B, \tilde{c}h) \rightarrow \mathcal{I}) \triangleright A & = & \text{svr}!(A, \tilde{c}h) \rightarrow (\mathcal{I} \triangleright A) \\
(\text{svr}(A, B, \tilde{c}h) \rightarrow \mathcal{I}) \triangleright B & = & \text{svr}?(B, \tilde{c}h) \rightarrow (\mathcal{I} \triangleright B) \\
(\text{svr}(A, B, \tilde{c}h) \rightarrow \mathcal{I}) \triangleright X & = & \mathcal{I} \triangleright X \quad - \text{if } X \notin \{A, B\} \\
(\text{ch}(A, B, \text{exp}) \rightarrow \mathcal{I}) \triangleright A & = & \text{ch}!(A, \text{exp}) \rightarrow (\mathcal{I} \triangleright A) \\
(\text{ch}(A, B, \text{exp}) \rightarrow \mathcal{I}) \triangleright B & = & \text{ch}?(B, \text{exp}) \rightarrow (\mathcal{I} \triangleright B) \\
(\text{ch}(A, B, \text{exp}) \rightarrow \mathcal{I}) \triangleright X & = & \mathcal{I} \triangleright X \quad - \text{if } X \notin \{A, B\} \\
(x := \text{exp}; \mathcal{I}) \triangleright X & = & x := \text{exp}; (\mathcal{I} \triangleright X) \quad - \text{if } X = \text{role}(x) \\
(x := \text{exp}; \mathcal{I}) \triangleright X & = & \mathcal{I} \triangleright X \quad - \text{if } X \neq \text{role}(x) \\
(\text{if } b \ \mathcal{I} \ \text{else} \ \mathcal{J}) \triangleright X & = & \text{if } b \ (\mathcal{I} \triangleright X) \ \text{else} \ (\mathcal{J} \triangleright X) \quad - \text{if } X = \text{role}(b) \\
(\text{if } b \ \mathcal{I} \ \text{else} \ \mathcal{J}) \triangleright X & = & (\mathcal{I} \triangleright X) \sqcup (\mathcal{J} \triangleright X) \quad - \text{if } X \neq \text{role}(b) \\
(\mathcal{I} \sqcup \mathcal{J}) \triangleright X & = & (\mathcal{I} \triangleright X) \sqcup (\mathcal{J} \triangleright X) \\
(\mathcal{I} \parallel \mathcal{J}) \triangleright X & = & (\mathcal{I} \triangleright X) \parallel (\mathcal{J} \triangleright X) \\
(\mathcal{I}; \mathcal{J}) \triangleright X & = & (\mathcal{I} \triangleright X); (\mathcal{J} \triangleright X)
\end{array}$$

Figure 3.4: Choreography to orchestration projection function

in [27], to check whether the choreography satisfies a sufficient condition for implementability, by syntactic analysis. We check whether the choreography is *strongly connected* (or *well threaded* [27]), which intuitively means whether there is sufficient communication between the service roles so that the choreography is implementable. In general, there are choreographies which are not strongly connected but implementable. Nevertheless, strongly-connectedness remains a desirable property. If a choreography is strongly connected, a sound prototype orchestration may be generated by projecting the relevant behaviors to the respective role. If the choreography is not strongly connected, we then offer to repair the choreography to make it strongly connected.

In the following, we present our approach in details. Firstly, we define a projection function which extracts relevant behaviors of a role from a choreography. Let  $\mathcal{I}$  be a choreography. The projection of  $\mathcal{I}$  onto role  $X$  is written as  $\mathcal{I} \triangleright X$ , which is defined by the rules presented in Fig. 3.4. We highlight that a conditional choice is projected to an unconditional choice if the condition is independent of variables associated with the role.

Ideally, given  $A, B, \dots, X$  as the roles  $\mathcal{I}$ ,  $\mathcal{I} \triangleright A \parallel \mathcal{I} \triangleright B \parallel \dots \parallel \mathcal{I} \triangleright X$  shall be trace-equivalent to

$$\begin{aligned}
\text{init}(\text{Stop}) &= \text{init}(\text{Skip}) = \emptyset \\
\text{init}(\text{svr}(A, B, \tilde{c}h) \rightarrow \mathcal{I}) &= \{A\} \\
\text{init}(\text{ch}(A, B, \text{exp}) \rightarrow \mathcal{I}) &= \{A\} \\
\text{init}(x := \text{exp}; \mathcal{I}) &= \{\text{role}(x)\} \\
\text{init}(\text{if } b \text{ } \mathcal{I} \text{ else } \mathcal{J}) &= \{\text{role}(b)\} \\
\text{init}(\mathcal{I} \square \mathcal{J}) &= \text{init}(\mathcal{I}) \cup \text{init}(\mathcal{J}) \\
\text{init}(\mathcal{I} \parallel \mathcal{J}) &= \text{init}(\mathcal{I}) \cup \text{init}(\mathcal{J}) \\
\text{init}(\mathcal{I}; \mathcal{J}) &= \text{init}(\mathcal{I})
\end{aligned}$$

Figure 3.5: Definition of Initiating Roles

$\mathcal{I}$ . This is not true for many reasons. For instance, assume that  $\mathcal{I}$  is as follows,  $\text{svr}(A, B, \tilde{c}h_1) \rightarrow \text{svr}(C, D, \tilde{c}h_2) \rightarrow \text{Stop}$ . It is easy to show that  $\mathcal{I} \triangleright A \parallel \mathcal{I} \triangleright B \parallel \mathcal{I} \triangleright C \parallel \mathcal{I} \triangleright D$  allows more behaviors than  $\mathcal{I}$  does. The reason is that the two interactions involve different roles and therefore it is impossible to ensure the global ordering without introducing extra communication. Another example is  $\mathcal{I}_{\text{exa}}$ , as shown above. In order to handle all choreographies and keep the synthesis algorithm simple, we take an alternative approach. We firstly define the sufficient conditions which guarantee the soundness of the projection and then discuss how to solve the problem if the conditions are not met.

**Definition 3.3.1** (Initiating roles). *Let  $\mathcal{I}$  be a choreography. The set of initiating roles of  $\mathcal{I}$ , written as  $\text{init}(\mathcal{I})$ , is defined in Figure 3.5. Similarly, we define the terminating roles of  $\mathcal{I}$ , written as  $\text{term}(\mathcal{I})$ , i.e., the roles participating in the last event of  $\mathcal{I}$ .*

**Definition 3.3.2** (Strongly-connectedness). *Let  $\mathcal{I}$  be a choreography.  $\mathcal{I}$  is strongly connected if and only if it can be inductively deduced from the following rules,*

- *Stop and Skip are strongly connected.*
- *$\text{svr}(A, B, \tilde{c}h) \rightarrow \mathcal{I}$  is strongly connected if and only if  $\text{init}(\mathcal{I}) = \{B\}$ , and  $\mathcal{I}$  is strongly connected.*
- *$\text{ch}(A, B, \text{exp}) \rightarrow \mathcal{I}$  is strongly connected if and only if  $\text{init}(\mathcal{I}) = \{B\}$ , and  $\mathcal{I}$  is strongly connected.*



- $x := \text{exp}; \mathcal{I}$  is strongly connected if and only if  $\{\text{role}(x)\} = \text{init}(\mathcal{I})$ , and  $\mathcal{I}$  is strongly connected.
- if  $b \ \mathcal{I} \ \text{else} \ \mathcal{J}$  is strongly connected if and only if both  $\mathcal{I}$  and  $\mathcal{J}$  are strongly connected, and  $\{\text{role}(b)\} = \text{init}(\mathcal{I}) = \text{init}(\mathcal{J})$ .
- $\mathcal{I} \square \mathcal{J}$  is strongly connected if and only if  $\text{init}(\mathcal{I}) = \text{init}(\mathcal{J})$ , and both  $\mathcal{I}$  and  $\mathcal{J}$  are strongly connected.
- $\mathcal{I} \parallel \mathcal{J}$  is strongly connected if and only if both  $\mathcal{I}$  and  $\mathcal{J}$  are strongly connected.
- $\mathcal{I}; \mathcal{J}$  is strongly connected if and only if both  $\mathcal{I}$  and  $\mathcal{J}$  are strongly connected and there exists role  $A$  such that  $\{A\} = \text{term}(\mathcal{I}) = \text{init}(\mathcal{J})$ .

Intuitively, a choreography is strongly connected if there is no “gap” between two consecutive statements. By definition, strongly connectedness can be checked syntactically and the complexity is linear in the size of the choreography. For instance, it is straightforward to verify that the choreography  $\mathcal{I}_{\text{exa}}$  (presented above) is not strongly connected because the two choices have different initiating roles. The choreography presented in Fig. 3.1 is not strongly connected because of the “gap” between the first two messages.

$$B2S(\text{Buyer}, \text{Seller}, \{\text{Bch}\})\text{Bch}(\text{Buyer}, \text{Seller}, \text{QuoteRequest})$$

The last role participated in the first message is *Seller*, whereas the initiating role of the second message is *Buyer*. We remark that if message sending/receiving is synchronous, then this choreography becomes “strongly connected”. This can be repaired by adding an acknowledge message from *Seller* to *Buyer* in between.

**Theorem 3.3.3.** *Let  $\mathcal{I}$  be a choreography. Let  $A, B, \dots, X$  be the roles participating in  $\mathcal{I}$ . Let  $\mathcal{O}$  be an orchestration such that  $\mathcal{O} = \mathcal{I} \triangleright A \parallel \mathcal{I} \triangleright B \parallel \dots \parallel \mathcal{I} \triangleright X$ . If  $\mathcal{I}$  is strongly connected, then  $\text{traces}(\mathcal{O}) = \text{traces}(\mathcal{I})$ .  $\square$*

$\mathcal{R}(\text{Stop}, S, E) = \text{Stop}$	-if $I$ is $\text{Stop}$ ;
$\mathcal{R}(\text{Skip}, S, E) = \text{Skip}$	-if $S = E$ ;
$\mathcal{R}(\text{Skip}, S, E) = \text{ch1}(S, E, *) \rightarrow \text{Skip}$	-if $S \neq E$ ;
$\mathcal{R}(\text{svr}(A, B, \tilde{c}h) \rightarrow I, S, E) = \text{svr}(A, B, \tilde{c}h) \rightarrow \mathcal{R}(I, B, E)$	-if $S = A$ ;
$\mathcal{R}(\text{svr}(A, B, \tilde{c}h) \rightarrow I, S, E) = \text{ch1}(S, A, *) \rightarrow \text{svr}(A, B, \tilde{c}h) \rightarrow \mathcal{R}(I, B, E)$	-if $S \neq A$ ;
$\mathcal{R}(\text{ch}(A, B, \text{exp}) \rightarrow I, S, E) = \text{ch}(A, B, \text{exp}) \rightarrow \mathcal{R}(I, B, E)$	-if $S = A$ ;
$\mathcal{R}(\text{ch}(A, B, \text{exp}) \rightarrow I, S, E) = \text{ch1}(S, A, *) \rightarrow \text{ch}(A, B, \text{exp}) \rightarrow \mathcal{R}(I, B, E)$	-if $S \neq A$ ;
$\mathcal{R}(x:=\text{exp}; I, S, E) = x:=\text{exp}; \mathcal{R}(I, S, E)$	-if $S = \text{role}(x)$ ;
$\mathcal{R}(x:=\text{exp}; I, S, E) = \text{ch1}(S, \text{role}(x), *) \rightarrow x:=\text{exp}; \mathcal{R}(I, S, E)$	-if $S \neq \text{role}(x)$ ;
$\mathcal{R}(\text{if } b \text{ } I \text{ else } J, S, E) = \text{if } b \text{ } \mathcal{R}(I, S, E) \text{ else } \mathcal{R}(J, S, E)$	-if $S = \text{role}(b)$ ;
$\mathcal{R}(\text{if } b \text{ } I \text{ else } J, S, E) = \text{ch1}(S, \text{role}(b), *) \rightarrow \text{if } b \text{ } \mathcal{R}(I, S, E) \text{ else } \mathcal{R}(J, S, E)$	-if $S \neq \text{role}(b)$ ;
$\mathcal{R}(I \square J, S, E) = \mathcal{R}(I, S, E) \square \mathcal{R}(J, S, E)$	-if $S \in \text{init}(I \square J)$
$\mathcal{R}(I \square J, S, E) = \text{ch1}(S, X, *) \rightarrow (\mathcal{R}(I, S, E) \square \mathcal{R}(J, S, E))$	-if $S \notin \text{init}(I \square J)$ and $X \in \text{init}(I \square J)$
$\mathcal{R}(I \parallel J, S, E) = \mathcal{R}(I, S, E) \parallel \mathcal{R}(J, S, E)$	-if $S \in \text{init}(I \parallel J)$
$\mathcal{R}(I \parallel J, S, E) = \text{ch1}(S, X, *) \rightarrow (\mathcal{R}(I, S, E) \parallel \mathcal{R}(J, S, E))$	-if $S \notin \text{init}(I \parallel J)$ and $X \in \text{init}(I \parallel J)$
$\mathcal{R}(I; J, S, E) = \mathcal{R}(I, S, X); \mathcal{R}(J, X, E)$	- $X \in \text{init}(J)$ .

Figure 3.6: Choreography repair function

The theorem states that strongly-connectedness serves as a sufficient condition for the correctness of the projection function presented in Fig. 3.4. It can be proved by structural induction. We skip the proof in this paper.

Strongly-connectedness allows to apply fully automated synthesis in a straightforward way. Nonetheless, because it requires all messages must be *connected*, e.g., a message output must be followed by an acknowledgement. Choreography drafts may not often be strongly connected. It is not very helpful if we simply claim a choreography is bad. Hence, we provide a method to automatically repair choreographies which are not strongly connected. The idea is to insert extra communications in order to fill the “gaps”.

Let  $S$  (for starting role),  $E$  (for ending role) be two roles. Let  $\mathcal{R}$  be the repairing function. Fig. 3.6 shows how  $\mathcal{R}$  calculates a refined choreography in a compositional way. Notice that we assume  $\text{ch1}$  is a channel between the sending role and receiving role and  $*$  is any message.

**Theorem 3.3.4.** *Let  $\mathcal{I}$  be an arbitrary choreography. Let  $start$  be a role in  $init(\mathcal{I})$ . Let  $end$  be a role in  $term(\mathcal{I})$ .  $\mathcal{R}(\mathcal{I}, start, end)$  is strongly connected.  $\square$*

The proof of the theorem is straightforward. By theorem 3.3.4, we may then apply the projection and generate a prototype orchestration. For instance, choreography  $\mathcal{I}_{exa}$  will be modified as follows,

$$\begin{aligned} &(ch(B, A, *) \rightarrow ch(A, B, M_1) \rightarrow ch(B, A, M_2) \rightarrow Stop) \\ &\quad \square(ch(B, A, M_1) \rightarrow ch(A, B, M_2) \rightarrow Stop) \end{aligned}$$

The following orchestration can then be generated

$$\begin{aligned} &((ch!* \rightarrow ch?M_1 \rightarrow ch!M_2 \rightarrow Stop) \\ &\quad \square(ch!M_1 \rightarrow ch?M_2 \rightarrow Stop))@B \\ &\parallel ((ch?* \rightarrow ch!M_1 \rightarrow ch?M_2 \rightarrow Stop) \\ &\quad \square(ch?M_1 \rightarrow ch!M_2 \rightarrow Stop))@A \end{aligned}$$

It can be shown that the generated orchestration above is equivalent to the repaired choreography. We remark it is wasteful to simply tell that a choreography is not implementable without telling how to correct it. *Our method gives the best effort to help users.* By comparing the repaired choreography and the original one, users are essentially presented why the original one is not implementable, and better, an easy way to correct it. In our toolkit, we offer other syntactic analysis as well, e.g., all kinds of well-formness. For instance, depending whether a channel is to be used once or multiple times (which can be specified in WS-CDL document), we can check whether a violation is possible during runtime.

In our formalism and Web-service composition languages such as WS-CDL, we may request the *service channel principle*. That is, service channels are intended to be repeatedly invocable and be always available to those who know the port names. Syntactically, this requires that  $inv?\tilde{x}$  shall not be preceded in every processes. The synthesized service, however, may not satisfy this principle. We perform a simple checking and give a warning message if the generated orchestration violates the principle. It is our future work to identify the

ways of generating orchestration which does satisfy the principle from a maximum set of choreographies.

### 3.4 Implementation and Evaluation

The methods discussed in previous sections have been realized in a toolkit named WS@PAT. WS@PAT is developed as a self-contained module in the PAT (Process Analysis Toolkit) framework, which is designed for supporting multiple domain specific modeling languages. WS@PAT has four main components, i.e., an editor with advanced editing features, a simulator which can be used to simulate the Web service models in different ways (e.g., interactive simulation, automated random simulation, generation of state graph, etc.), a verifier which integrates different model checking algorithms for different properties and a synthesizer which performs choreography repairing and orchestration generation. WS@PAT has been applied to multiple case studies, including ones from <http://www.oracle.com/> and from [27, 78]. We are currently applying WS@PAT to several large WS-CDL and WS-BPEL models. Notice that our approach for synthesis is based on syntactic analysis and therefore scales up for large Web service models. We thus demonstrate the scalability of our verification approach, using two models. One is the online store example presented in Fig. 3.1 and Fig. 3.3. Instead of one buyer and one service invocation, we amend the model so that multiple users are allowed to use the services multiple times. The other is the service for travel arrangement. Its WS-BPEL model is available at <http://www.comp.nus.edu.sg/~pat/cdl/>. A WS-CDL specification is created manually. A number of clients invoke the business process, specifying the name of the employee, the destination, the departure date, and the return date. The BPEL process checks the employee travel status (through a Web service). Then it checks the prices for the flight ticket with multiple airlines (through Web services). Finally, the BPEL process selects the lowest price and returns the travel plan to each client.

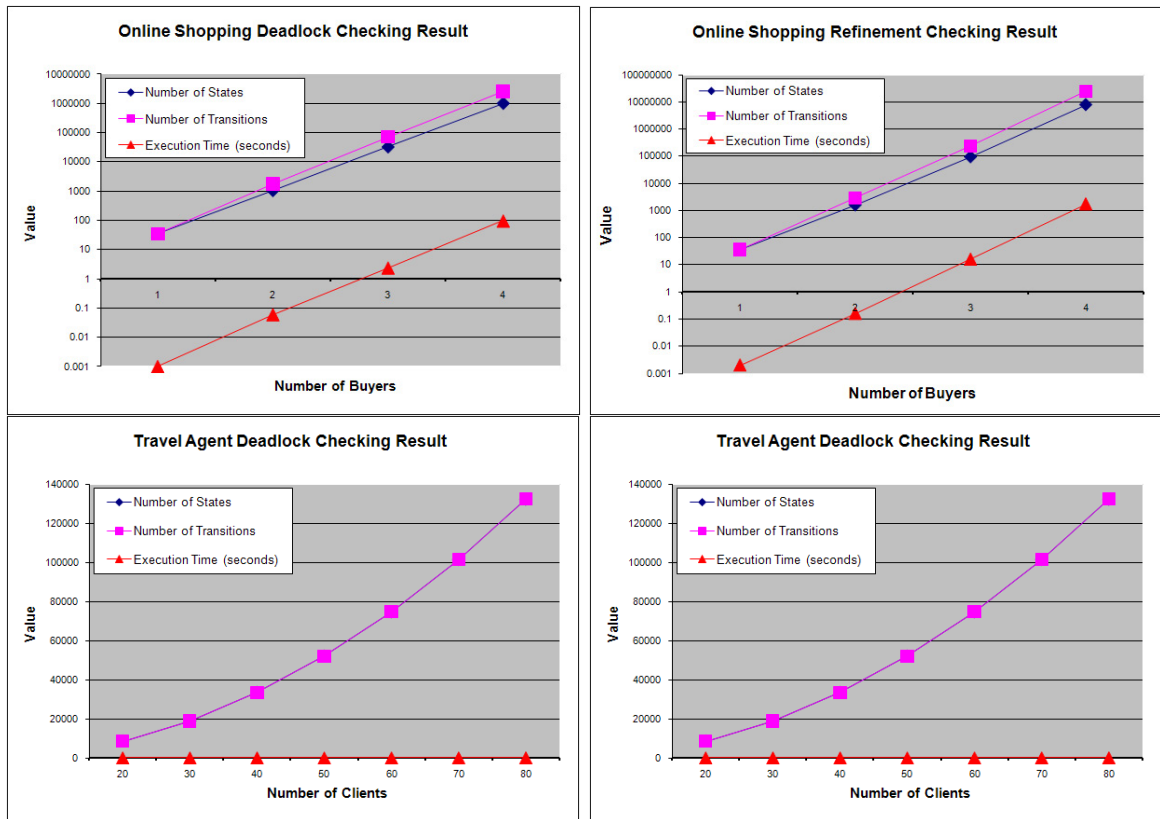


Figure 3.7: WS@PAT verification performance

Fig. 3.7 shows WS@PAT’s efficiency using the two examples, obtained on a PC with Intel Q9500 CPU at 2.83GHz and 4GB RAM. Notice that in the experiments, we model the physical constraints as in [43] and verify the whole system instead of one service invocation. For both examples, we verify whether the orchestration is deadlock-free or not, by a reachability analysis searching for a deadlock state. In the online store example, we allow buyers to invoke the service repeatedly. As a result, the orchestration is deadlock-free. In the travel arrangement example, one client invokes the service only once. Because the number of concurrent service invocations is bound by the maximum number of threads allowed, the system reaches a deadlock state after exhausting all threads. This is consistent with the finding in [43]. In such case, WS@PAT is able to find a counterexample execution reasonably quickly with 80 clients using the service at the same time. We also verify that

the orchestration conforms to the choreography using the refinement checking algorithm, as shown in Fig. 3.7. In both cases, the number of states and the time increase rapidly. Yet, WS@PAT is able to confirm that the orchestration conforms to the choreography with a few buyers/clients using the service concurrently.

In a nutshell, WS@PAT explores  $10^8$  states in a few hours, which suggests that WS@PAT is comparable to FDR [79] and SPIN [54] in terms of efficiency. WS@PAT shares many ideas with WS-Engineer. To compare with WS-Engineer, we use Police Enquiry Obligations case study inside WS-Engineer. In the original example, the police officer will send request to a officer device, and following that, the officer device will enquire some items in sequence, e.g., nominal record, vehicle record, insurance record, ANPR record, DNA record, and then reply to the officer for the information. In order to make the example more challenging, we let the officer device enquire items in parallel. The size in the table above denotes the number of items to be retrieved in parallel. For each size, we provide two orchestrations, one conforms to the choreography (*Correct Police*), while the other one replies to the police officer before retrieving the items (*Wrong Police*).

The experiments data in the Table 3.3 below show that WS@PAT is faster than WS-Engineer for *Correct Police* cases. When the conformance does not hold, WS@PAT stops immediately after the counterexample is detected, but not for WS-Engineer. We conclude that WS@PAT complements WS-Engineer for the orchestration synthesis, and still has competitive performance. The result, however, should be taken with a grain of salt, since both the input languages and verification algorithms are different.

### 3.5 Related Work

The work is related to research on verifying or synthesizing Web services [48, 23, 22], particularly, the line of work by Foster *et al* presented in [44, 45, 43]. They proposed to

Example		WS@PAT		WS-Engineer	
Name	Size	Time	#States	Time	#States
Correct Police	6	0.25	734	0.3	731
Wrong Police	6	0.02	3	0.4	731
Correct Police	7	0.93	2192	1.1	2189
Wrong Police	7	0.02	3	1	2189
Correct Police	8	3.96	6566	6.2	6563
Wrong Police	8	0.02	3	6.2	6563
Correct Police	9	15.57	19688	51.3	19685
Wrong Police	9	0.02	3	50.7	19685

Table 3.3: WS@PAT vs WS-Engineer

apply model-based verification for Web services. Their approach is to build Finite State Processes (FSP) model from Web services and then apply verification techniques based on FSP to verify Web services. For instance, conformance between choreography and orchestration is verified by showing a bi-simulation relationship between the respective FSP models. In particular, they identified the model of resource constraint in Web service verification [43] and proposed to perform verification under resource constraints. In addition, they developed a tool named LTSA-WS [45] (and later WS-Engineer). Our work can also be categorized as model-based verification, and is similar to theirs. Our approach complements their works in a number of aspects. Firstly, our model is based on a modeling language which is specially designed for Web service composition with features like channel passing, shared variables/arrays, service invocation with service replication, etc. Secondly, our verification algorithms employ specialized optimizations for Web services verification, e.g., model reduction based on algebraic properties of the models, partial order reduction for orchestration with multiple local computational steps, etc. These optimizations allow us to handle large state space and potentially large Web services. Lastly, we study the synthesis problem and offer a lightweight and practical solution, which is related to the work presented in [26]. The synthesis approach in [22] generates high level behavior patterns from WSDL description, while our approach synthesizes implementation from

WS-CDL design.

The conformance checking is also discussed in [58, 19, 7]. In [58], formalizations are provided for the two views and symbolic model checking is used for the conformance checking. In [19] the notion of conformance is defined by means of automata and is restricted only to compositions of two services. The work of [7] concentrates on checking that the choreography specification is respected by the implementing services at run time. The formalization is given in terms of Petri Nets. Compared with these approaches, we provide a unified semantic model for Web service composition with efficient verification algorithm.

This work is related to works on verifying WS-BPEL [39, 57] and WS-CDL [76] by translating to other formalisms and verifying using existing model checkers like Uppaal [39], Java Path Finder [76] or NuSMV [57]. Compared to them, we provide direct verification and dedicated optimizations for the Web services specification languages. Our approach follows the formalization of Web service and the discussion on Web service generation in [27] and [78]. Our modeling languages are inspired from the simple Web service languages used in [27, 78] (which are sufficient for theoretical discussion). However, in order to develop a useful tool, we extend them to cover larger subset of the language constructs for Web service compositions. For example, variables and channel messages are supported in our languages but abstracted out in [78], which makes the modeling of real-world systems much easier. One could argue that it is possible to model Web services using other process algebra like modeling language, like Promela, or MSC for choreography and FSM for orchestration. These proposals suffer from the disadvantages of the translation approach. For example, the translation from Web service model to target process algebra may not be optimal, and the reflection of the counterexample is also non-trivial. Additionally, specific verification may not supported is the existing tool. For instance, the SPIN model checker for Promela does not support refinement checking, hence it is not possible



to check the Web service conformance. WS@PAT as a verifier is related to tools on equivalence/refinement checking (or language containment checking), e.g., FDR. Motivated by the features of Web services, we extend the algorithms to check conformance relations with specialized optimizations.

## Chapter 4

# Verification with Compositional Partial Order Reduction

Web services have emerged as an important technology nowadays, and Web services support much concurrency. With the advent of multi-core and multi-CPU systems, concurrent systems are widely used nowadays. Nevertheless, it is not a simple task for programmers to maximize the benefit of concurrency, as programmers are often burdened with the task of handling threads and locks explicitly. In addition, it is favorable that processes can be composed at different granularity level, from simple processes to complete workflows. *Orc* calculus [60] is designed to specify orchestrations and wide-area computations in a concise and structured manner. Albeit it only has a few concurrency combinators, it can manage timeouts, priorities, and failures effectively [60]. To fully utilize the multi-processor systems, operational semantics [93, 59] of *Orc* allows the executions to run in parallel whenever it is possible. Since concurrency bugs are difficult to discover solely by testing, therefore it is desirable to verify the language formally. Nevertheless, semantics of *Orc* that is highly concurrent has made states grow exponentially and posed a challenge for its formal verification.

In the literature, various state reduction techniques have been proposed to tackle the problem, successful ones including on-the-fly verification [53], symmetry reduction [33, 41], partial order reduction (POR) [60, 73, 50, 92, 18, 74], etc. POR works by exploiting the independency of concurrently executed transitions in order to reduce the number of possible interleaving. For example, consider the transition system in Figure 4.1,  $t_1$  and  $t_2$  are independent transitions, which means that executing either  $t_1t_2$  or  $t_2t_1$  from a state  $s_1$  will always lead to the same state  $s_2$ . POR will detect such independency, and choose only  $t_1t_2$  for execution, thus reduce the state-space of exploration. Common POR algorithms, such as [92, 50, 60, 73, 18], works by identifying a subset of outgoing transitions of a state that is sufficient for verification.

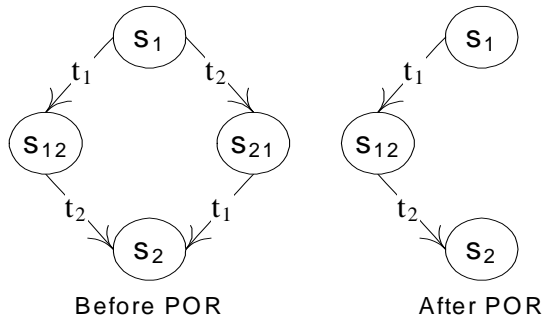


Figure 4.1: Partial Order Reduction

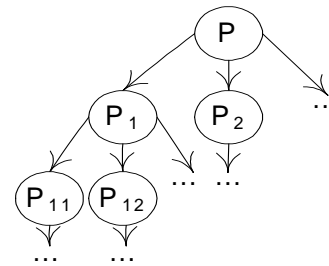


Figure 4.2: Hierarchical Concurrent Processes

Nowadays, many concurrent systems are designed in top-down architecture, and concurrent processes are structured in a hierarchical way as shown in Figure 4.2. In figure 4.2, process  $P$  contains sub processes  $P_1, P_2$ , etc that are running in parallel. Process  $P_1$  in turn contains sub processes  $P_{11}, P_{12}$ , etc that are running in parallel. We refer concurrent processes of this kind as hierarchical concurrent processes (HCP). There are many real-life examples of such. Considering a browser that supports tabbed browsing, multiple instances of browser windows could be opened at the same time, and each browser window could contain multiple opened tabs, and each opened tab could download several HTML

elements in parallel. An *Orc* process could be viewed as a process that is composed by HCP.

Current POR algorithms have a common assumptions that *local transitions* within the participated processes are dependent to each other. In the context of HCP as shown in Figure 4.2, if POR is applied on process  $P$ , transitions within processes  $P_1, P_2$  will be considered as *local transitions*, therefore they are simply assumed to be dependent. Nevertheless, the fact is that many of the local transitions are indeed independent. In this work, we propose a method called Compositional Partial Order Reduction (CPOR), which extends POR to the context of HCP. Different from current POR methods, CPOR further exploits the independency within the local transitions. CPOR works by applying POR recursively for the hierarchical concurrent processes, and several possible ample sets are composed in a bottom-up manner. In order to apply CPOR in *Orc* language, we first define the structure of HCP of an *Orc* process. Subsequently, base on the structure of HCP of an *Orc* process, we established some local criteria that could be easily checked by CPOR algorithm. Experimental results show that CPOR has greatly reduced the state space of verification of *Orc* language.

**Chapter Outline.** Section 4.1 introduces *Orc* language. Section 4.2 elaborates on CPOR and shows how it can be applied to *Orc* models. Section 4.3 gives several experimental results. Section 4.4 surveys the related work.

## 4.1 Orchestration Language Orc

### 4.1.1 Syntax

*Orc* is a service orchestration language in which multiple services are invoked to achieve a goal while managing time-outs, priorities, and failures of services or communication.

Variable	$x$	::=	variable name	
Value	$m$	::=	value	
Parameter	$p$	::=	$x \mid m$	
Expression	$E$	::=	$M(\bar{p})$	site call
			$F(\bar{p})$	function call
			$E \mid E$	parallel
			$E > x > E$	sequential
			$E < x < E$	pruning
			$E ; E$	otherwise

Figure 4.3: Syntax of Orc

Figure 4.3 describes the syntax of *Orc*:

**Site** The simplest *Orc* expression is a site call  $M(\bar{p})$ , where  $M$  is the service’s name and  $\bar{p}$  is a list of parameters. Sites are basic units of *Orc* language. A site can be an external service (e.g. *Google* site) which resides on different machine. For example,  $Google("Orc")$  is an external site call that calls the external service provided by *Google* and its response is the search results for keyword “*Orc*” by *Google* search engine. A site can also be a local service (e.g. *plus* site) which resides on the same machine. For example,  $plus(1,1)$  is a local site call that calls the local *plus* service and its response is the summation of two arguments. Since site in *Orc* is essentially a service, henceforth, we would use the term *site* and *service* interchangeably.

**Combinators** There are four combinators: parallel, sequential, pruning, and otherwise combinators. The parallel combinator  $F \mid G$  defines a parallel expression, where expressions  $F$  and  $G$  execute independently, and its published value can be the value published either by  $F$  or by  $G$  or both of them. The sequential combinator  $F > x > G$  defines a sequential expression, where each value published by  $F$  initiates a separate execution of  $G$  wherein  $x$  is bound to it. The execution of  $F$  is then continued in parallel with all these executions of  $G$ . The values published by the sequential expression are the values published by the executions of  $G$ . For example,  $(Google("Orc") \mid Yahoo("Orc")) > x > Email(addr, x)$

will call Google and Yahoo sites simultaneously. For each returned value, an instance of  $x$  will be bound to it, and an email will be sent to  $addr$  for each instance of  $x$ . Thus, up to two emails will be sent. If  $x$  is not used in  $G$ ,  $F \gg G$  can be used as a shorthand for  $F > x > G$ . The pruning combinator  $F < x < G$  defines a pruning expression, where initially  $F$  and  $G$  execute in parallel. However, when  $F$  needs the value of  $x$ , it will be blocked until  $G$  publishes a value to bind  $x$  and  $G$  terminates immediately after that. For example,  $Email(addr, x) < x < (Google("Orc")|Yahoo("Orc"))$  will get the fastest searching results for the email sending to  $addr$ . In contrast to sequential expressions, it will publish at most one value. If  $x$  is not used in  $F$ ,  $F \ll G$  can be used as a shorthand for  $F < x < G$ . The otherwise combinator  $F ; G$  defines an otherwise expression, where  $F$  executes first. The execution of  $F$  is replaced by  $G$  if  $F$  halts without any published value.  $F$  halts if all site calls are responded or halted, and furthermore, it does not publish any more values or call any more sites.

**Functional Core Language (Cor)** *Orc* is enhanced with functional core language (Cor) to support various data types, mathematical operators, conditional expressions, function calls, etc. Cor structures such as conditional expressions and functions are translated into site calls and four combinators [60]. For example, conditional expression *if E then F else G*, where  $E, F, G$  are *Orc* expressions would be translated into expression  $(if(b) \gg F | if(\sim b) \gg G) < b < E$  before evaluation.

### Example - Metronome

Timer is explicitly supported by *Orc* by introducing time-related sites that delay a given amount of time. One of such sites is *Rtimer*, for example,  $Rtimer(5000) \gg "Orc"$  will publish "Orc" at 5 seconds. Like most other programming languages, Cor provides the capability to define functions, which are expressions that have a defined name, and have some number

of parameters, function are declared using the keyword *def*. Following is a function that defines a metronome [60], which will publish a *signal* value every *t* seconds. *signal* is a value that carries no information. Note that the function is defined recursively.

$$\text{def } \text{metronome}(t) = (\text{signal} \mid \text{Rtimer}(t) \gg \text{metronome}(t))$$

The following example publishes “tick” once per second, and publishes “tock” once per second after an initial half-second delay.

$$(\text{metronome}(1000) \gg \text{“tick”}) \mid (\text{Rtimer}(500) \gg \text{metronome}(1000) \gg \text{“tock”})$$

Thus the publications are “tick tock tick ...” where “tick” and “tock” alternate each other. One of the properties that we are interested is whether the system could publish two consecutive “tick”s or two consecutive “tock”s which is an undesirable situation. In order to easily assert a global property that holds throughout the execution of an *Orc* program, we extend *Orc* with auxiliary variables. The value of an auxiliary variable could be accessed and updated throughout the *Orc* program. Henceforth, we will simply refer the extended auxiliary variables simply as *global variables*. A global variable is declared with the keyword *globalvar* and a special site, *\$GUpdate*, is used to perform update on a global variable. Consider the metronome example with a global variable *tickNum* and site *\$GUpdate* inserted,

$$\begin{aligned} &\text{globalvar } \text{tickNum} = 0 \\ &\text{def } \text{metronome}(t) = (\text{signal} \mid \text{Rtimer}(t) \gg \text{metronome}(t)) \\ &(\text{metronome}(1000) \gg \$\text{GUpdate}(\{\text{tickNum} = \text{tickNum} + 1\}) \gg \text{“tick”}) \\ &\mid (\text{Rtimer}(500) \gg \text{metronome}(1000) \gg \$\text{GUpdate}(\{\text{tickNum} = \text{tickNum} - 1\}) \\ &\gg \text{“tock”}) \end{aligned}$$

We can verify whether it is possible to have two consecutive “tick”s or two consecutive “tock”s by checking whether the system is able to reach a state satisfying the condition  $(\text{tickNum} < 0 \vee \text{tickNum} > 1)$ .

### 4.1.2 Semantics

This section presents the semantic model of *Orc* based on Label Transition System (LTS). The timed operational semantics of *Orc* can be found in [93, 59]. In the following, we introduce some definitions required in semantic model.

**Definition 4.1.1** (System Configuration). *A system configuration contains two components  $(Proc, Val)$ , where  $Proc$  is a process expression of *Orc*, and  $Val$  is a valuation function, which maps the variables to their values.*

A variable in the system could be an *Orc*'s variable, or the global variable which is introduced for capturing global properties. The value of a variable could be a primitive value, a reference to a site, or a state object. The three primitive types supported by *Orc* are boolean, integer, and string. All variables are assumed to have finite domain. Two configurations are equivalent iff they have the same process expression  $Proc$  and same valuation function  $Val$ . Proc component of system configuration is assumed to have finitely many values.

**Definition 4.1.2** (System Model). *A system model is a 3-tuple  $\mathcal{S} = (Var, init_G, P)$ , where  $Var$  is a finite set of global variables,  $init_G$  is the initial (partial) variable valuation function and  $P$  is the *Orc* expression.*

**Definition 4.1.3** (System Action). *A system action contains four components  $(Event, Time, EnableSiteType, EnableSiteId)$ .  $Event$  is either publication event, written  $!m$  or internal event, written  $\tau$ .  $EnableSiteType, EnableSiteId$  are the type and unique identity of the site that initiates the system action.  $Time$  is the total delay time in system configuration before the system action is triggered.*

Every system action is initiated by a site call, and we extend the system action defined in [93] with two additional components,  $EnableSiteType$  and  $EnableSiteId$ , to provide information for CPOR. A publication event  $!m$  communicates with the environment with value  $m$ , while an internal event  $\tau$  is invisible to the environment. There are three groups of site calls. The



first two groups are site calls for stateless and stateful services respectively. And the third are the site calls for  $\$GUpdate$  which update global variables. These three groups are denoted as *stateless*, *stateful*, and *GUpdate* respectively, and those are the possible values for *EnableSiteType*. Every site in the system model is assigned a unique identity which ranges over non-negative integer value. Discrete time semantics [93] is assumed in the system. *Time* ranges over non-negative integer value and is assumed to have finite domains.

**Definition 4.1.4** (Labeled Transition System (LTS)). *Given a model  $\mathcal{S} = (Var, init_G, P)$ , let  $\Sigma$  denote the set of system actions in  $P$ . The LTS corresponding to  $\mathcal{S}$  is a 3-tuple  $(C, init, \rightarrow)$ , where  $C$  is the set of all configurations,  $init \in C$  is the initial system configuration  $(P, init_G)$ , and  $\rightarrow \subseteq C \times \Sigma \times C$  is a labeled transition relation, and its definition is according to the operational semantics of *Orc* [93].*

To improve readability, we use  $c \xrightarrow{a} c'$  for  $(c, a, c') \in \rightarrow$ . An action  $a \in \Sigma$  is enabled in a configuration  $c \in C$ , denoted as  $c \xrightarrow{a}$ , iff there exists a configuration  $c' \in C$ , such that  $c \xrightarrow{a} c'$ . An action  $a \in \Sigma$  is *disabled* in a configuration  $c = (P, V)$ , where  $c \in C$ , iff the action  $a$  is not enabled in the configuration  $c$ , but it is enabled in some configurations  $(P, V')$ , where  $V' \neq V$ .  $Act(c)$  is used to denote the set of enabled actions of configuration  $c$ .  $Enable(c, a)$  is used to denote the set of reachable configurations through action  $a$  from configuration  $c \in C$ , formally, for any  $c \in C$ ,  $Act(c) = \{a \in \Sigma | c \xrightarrow{a}\}$ .  $Enable(c)$  is used to denote the set of reachable configurations from configuration  $c \in C$ , that is, for any  $c \in C$ ,  $Enable(c) = \{c' \in Enable(c, a) | a \in \Sigma\}$ .  $Ample(c)$  is used to denote the ample set (Section 4.2) of configuration  $c \in C$ .  $AmpleAct(c)$  is defined as the set of actions that caused a configuration  $c \in C$  transit into the configurations in  $Ample(c)$ , that is, for any  $c \in C$ ,  $AmpleAct(c) = \{a \in \Sigma | c \xrightarrow{a} c', c' \in Ample(c)\}$ .  $PAct(c)$  is used to denote the set of enabled and disabled actions of configuration  $c$ , and  $Act(c) \subseteq PAct(c)$ . We use  $TS$  to represent the original LTS before POR is applied and  $\widehat{TS}$  to represent the reduced LTS after POR is applied.  $TS_c$  is used to represent the LTS (before any reduction) that starts from configuration  $c$ . An

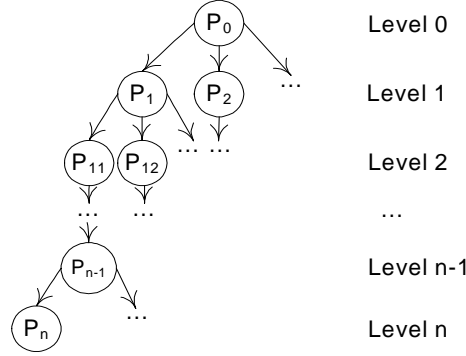


Figure 4.4: HCP of a general hierarchical concurrent process

execution fragment  $l = c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \dots$  of LTS is an alternating sequence of configurations and actions. A *finite execution fragment* is an execution fragment ending with a configuration.

We are interested in checking the system against two kinds of properties. The first kind is deadlock-freeness, which is to check whether there does not exist a configuration  $c \in C$  in TS such that  $Enable(c) = \emptyset$ . The second kind is temporal properties that are expressible with LTL without Next Operator (LTL-X) [34]. For any LTL-X formula  $\phi$ ,  $prop(\phi)$  denotes the set of atomic propositions used in  $\phi$ . In the metronome example which augmented with a global variable  $tickNum$ ,  $prop(\phi) = \{(tickNum < 0); (tickNum > 1)\}$ . An action  $a \in \sigma$  is  $\phi$ -invisible iff the action does not change the values of propositions in  $prop(\phi)$  for all  $c \in C$  in TS.

### 4.1.3 Hierarchical Concurrent Processes (HCP)

The general structure of a hierarchical concurrent process  $P$  is shown graphically using a tree structure in Figure 4.4. Henceforth, we denote such graph as HCP graph, or simply HCP if it does not lead to ambiguity.

Figure 4.4 shows that process  $P_0$  contains sub processes  $P_1, P_2$ , etc that are running concurrently. Process  $P_1$  in turn contains sub processes  $P_{11}, P_{12}$ , etc that are running concurrently.

This goes repeatedly until reaching a process  $P_n$  which has no subprocesses. Each process  $P$  in the hierarchy will have its associated level, starting from level 0. A process without any subprocess (e.g. process  $P_n$ ) is denoted as *terminal process*, otherwise the process is denoted as *non-terminal process*. Furthermore, process  $P_0$  at level 0 is denoted as *global process*, while processes at level  $i$ , where  $i > 0$ , are denoted as *local processes*. The *parent process* of a local process  $P'$  is a unique process  $P$  such that there is a directed edge from  $P$  to  $P'$  in the HCP graph. When  $P$  is the parent process of  $P'$ ,  $P'$  is called the *child process* of  $P$ . *Ancestor processes* of a local process  $P'$  are the processes in the path from global process to  $P'$ . *Descendant processes* of process  $P$  are those local processes that have  $P$  as an *ancestor process*. An *Orc* expression  $P$  could be viewed as a process that is composed by HCP. This could be formalized by constructing the HCP according to syntax of  $P$ , assigning process identity to each sub-expression of  $P$ , and defining how the defined processes evolve during the execution of expression  $P$ . In the following, we illustrate this in detail. An *Orc* expression can be either a site call or one of the four combinators and their corresponding HCPs are shown in Figure 4.5. A site call is a terminal process node, while each of the combinators has either one or two child processes according to their semantics (refer to Section 4.2), and the HCPs of respective child process nodes are defined recursively. We denote expressions  $A$  and  $B$  as LHS process and RHS process for each combinators in Figure 4.5. For example, a pruning combinator ( $A < x < B$ ) contains two child nodes because its LHS process and RHS process could be executed concurrently. Each of the process nodes in HCP is identified by a unique process identity (*pid*), and node values in HCP are prefixed with their *pid* (e.g.  $p_0, p_1$ , etc.). In Figure 4.6, an expression  $(S_1 \ll S_2)|(S_3 \ll S_4)$ , where  $S_1, S_2, S_3$ , and  $S_4$  are site calls, could be viewed as a process composed by HCP of three levels.

Consider a transition  $(P, V) \xrightarrow{a} (P', V')$ , where  $a$  is some action. We abuse the notation by using  $P$  and  $P_0$  to denote the HCPs before and after the transition. In fact,  $P_0$  could have different tree structures from  $P$ , and processes could be added or deleted in  $P_0$ . In order

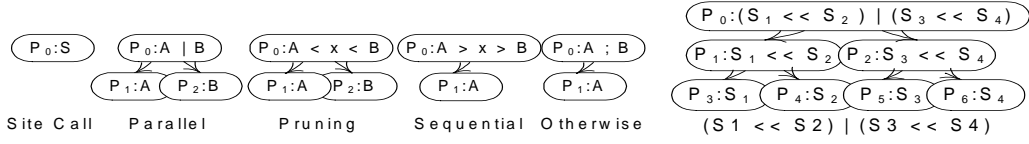


Figure 4.5: HCP of General Orc Expressions

Figure 4.6: An Orc Example

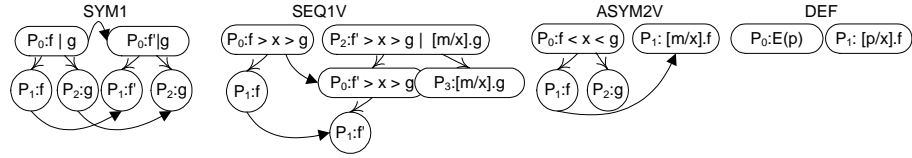


Figure 4.7: Relation of Processes between  $P$  and  $P'$

to have a clear relation of processes between  $P$  and  $P_0$ , we define the relation of processes between  $P$  and  $P_0$  over each rule of the operational semantics of *Orc* [89], some of which are presented in Figure 4.7 for illustration purpose. There are two HCPs under each rule. HCPs on the left and right are the HCPs before and after triggering the action initiated by respective rules. Two process nodes on different HCPs belong to the same process if they have the same *pid* value, and an arrow is used to relate them. Processes that could only be found in HCP on the right or left are the processes that are newly added or deleted respectively. In SEQ1V, the transition of  $f$  to  $f'$  produces an output value  $m$ , and notation  $[m/x].g$  is used to denote that all the instances of variable  $x$  in  $g$  are replaced with value  $m$ .

A site  $S$  is *private* in  $P_1[P]$ , if the reference of site  $S$  could not be accessed by all processes other than process  $P_1$  and its descendant processes under HCP graph of global process  $P$ . Otherwise, site  $s$  is *shared* in process  $P_1[P]$ . A site  $S$  is *permanently private* in  $P_1[c]$ , if for any configuration  $c' = (P', V')$  that is reachable by  $c$ , if  $P'$  has  $P_1$  as its descendant process, site  $S$  must be private in process  $P_1[P']$ .

The example in Figure 4.8 shows an *Orc* process  $P = A|B$ . Variables *userdb* and *flightdb* will be initialized to different instances of site *Buffer*, which provides the service of FIFO queue. In process  $A$ , two string values *user1* and *user2* are enqueued in the buffer referenced by *userdb*

$$A = (userdb : put("user1") \mid userdb : put("user2")) < userdb < Buffer()$$

$$B = (flightdb : put("CX510") \mid flightdb : put("CX511")) < flightdb < Buffer()$$
Figure 4.8: Execution of Orc process  $P = A \mid B$ 

concurrently. Buffer site that is referenced by *userdb* is *private* in  $A[P]$ , since *userdb* could only be accessed by process A. Now consider at some level  $j$  of HCP graph of global process  $P$ , where  $j > 1$ , we have processes  $P_{j_1} = userdb.put("user1")$  and  $P_{j_2} = userdb.put("user2")$ . *Buffer* site that is referenced by *userdb* is *shared* in  $P_{j_1}[P]$ , since *userdb* could be accessed by  $P_{j_2}$  which is not a descendant process of  $P_{j_1}$ .

## 4.2 Compositional Partial Order Reduction (CPOR)

The aim of Partial Order Reduction (POR) is to reduce the number of possible orderings by fixing the order of independent transitions as shown in Figure 4.1. The notion of *independency* plays a central role in POR, which is defined below by following [51].

**Definition 4.2.1** (Independency). *Two actions  $a_1$  and  $a_2$  in an LTS are independent if for any configuration  $c$  such that  $a_1, a_2 \in Act(c)$ :*

1.  $a_2 \in Act(c_1)$  where  $c_1 \in Enable(c, a_1)$  and  $a_1 \in Act(c_2)$  where  $c_2 \in Enable(c, a_2)$ ,
2. Starting from  $c$ , any configuration reachable by executing  $a_1$  followed by  $a_2$ , can also be reached by executing  $a_2$  followed by  $a_1$ .

*Otherwise, two actions are dependent.*

Given a configuration, an ample set is a subset of outgoing transitions of the configuration which are sufficient for verification, and it is formally defined as follow:

**Definition 4.2.2** (Ample Set). *Given an LTL-X property  $\emptyset$ , and a configuration  $c \in C$  in TS, an ample set is a subset of the enable set which must satisfy the following conditions [18]:*

**(A1) Nonemptiness condition:**  $Ample(c) = \emptyset$  if and only if  $Enable(c) = \emptyset$ .

**(A2) Dependency condition:** Let  $c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} c_n \xrightarrow{a} t$  be a finite execution fragment in  $TS$ . If  $a$  depends on some actions in  $AmpleAct(c_0)$ , then  $a_i \in AmpleAct(c_0)$  for some  $0 < i \leq n$ .

**(A3) Stutter condition:** If  $Ample(c) \neq Enable(c)$ , then any  $\alpha \in AmpleAct(c)$  is an invisible action.

**(A4) Strong Cycle condition:** Any cycle in  $\widehat{TS}$  contains at least one configuration  $c$  with  $Ample(c) = Enable(c)$ .

To be specific, reduced LTS generated by the ample set approach needs to satisfy conditions A1 to A4 in order to preserve the checking of LTL-X properties. However, for the checking of deadlock-freeness, only conditions A1 and A2 are needed [50]. Henceforth, our discussion will be focused on the checking of LTL-X property, but the reader could adjust accordingly for the checking of deadlock-freeness.

Conditions A1, A3, and A4 are relatively easy to check, while condition A2 is the most challenging condition. It is known that checking condition A2 is equivalent to checking the reachability of a condition in the full transition system  $TS$  [18]. It is desirable that we could have an alternative condition  $A2'$  that only imposes requirements on the current configuration instead of all traces in  $TS$ , and satisfaction of condition  $A2'$  would guarantee the satisfaction of condition A2. Given a configuration  $c_g = (P_g, V_g)$ , and  $P_d$  as a descendant process of  $P_g$ , with associated configuration  $c_d = (P_d, V_d)$ , we define a condition  $A2'$  that based solely on  $c_d$ , and its soundness will be proved in Section 4.2.3.

**(A2') Local Criteria of A2:** For all configurations  $c_a \in Ample(c_d)$  and  $c_a = (p_a, v_a)$  the following two conditions must be satisfied:

- (1) The enable site for the action  $a$  that enable  $c_a$  must be either stateless site, or stateful site private in  $p_a[P_g]$ ;
- (2)  $p_a$  is not a descendant process of the RHS process of some pruning combinators or the LHS process of some sequential combinators.

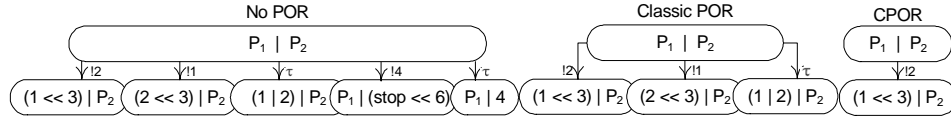


Figure 4.9: LTS of *Orc* Process  $P = (P_1 | P_2)$ ,  $P_1 = ((1 | 2) \ll 3)$ ,  $P_2 = (4 \ll 6)$

Notice that we define an ample set as a set of enabled configurations rather than a set of enabled actions like [18]. The reason is due to in references like [18], action-deterministic system is assumed. This entails that for any configuration  $c \in C$  and any action  $a \in \Sigma$ ,  $c$  has at most one outgoing transition with action  $a$ , formally,  $c \xrightarrow{a} c'$  and  $c \xrightarrow{a} c''$  implies  $c' = c''$ . Therefore, the enabled configurations could be deduced by the enabled actions. Nonetheless, an *Orc* system is not action-deterministic, the main reason is because some events in *Orc* are internal events that are invisible to the environment. By defining ample set as a set of configurations, with their associated enabled actions, the requirement of action-deterministic system is no longer needed.

### 4.2.1 Classic POR and CPOR

Classic POR methods assume that local transitions of a process are dependent, and in the context of HCP, it means that actions within individual processes from level 1 onwards are *simply assumed to be dependent*.

In Figure 4.9, three LTSs of the process  $P$  are given. No POR shows the set of all initial transitions of process  $P$ ; classic POR shows how the state-space of a parallel composition can be reduced when its component processes are independent; and CPOR reduces the initial actions further by examining internal process structure. For simplicity, system configuration is represented only by process expression. When no POR is applied, all interleavings of transitions are considered, and there are five branches after the initial state. When the classic POR is applied, since  $P_1$  and  $P_2$  are active processes, assume that it checks

process  $P_1$  first. All transitions of  $P_1$  are assumed to be dependent by the classic POR. For this reason the resulting ample set of  $P$  is  $\{(1 \ll 3) | P_2; (2 \ll 3) | P_2; ((1 | 2) | P_2)\}$ , which is a valid ample set after checking for conditions A1-A4. Therefore, there are three branches from initial state when classic POR is applied. Different from classic POR, when CPOR is applied, POR is again applied to process  $(1 | 2)$ . We define  $Ample(P)$  as a set of ample sets of process  $P$  that satisfy conditions A1 and A2, but yet to be checked for conditions A3 and A4.  $Ample((1 | 2)) = \{\{1\}; \{2\}\}$  and  $Ample(P_1)$  is  $Ample((1 | 2))$  after restructuring by the semantics of  $P_1$ , which is  $\{1 \ll 3; 2 \ll 3\}$ .  $Ample(P)$  is  $Ample(P_1)$  after restructuring by the semantics of  $P$ , which is  $\{1 \ll 3 | P_2, 2 \ll 3 | P_2\}$ . Each ample set in  $Ample(P)$  will then be checked for conditions A3 and A4, and both ample sets turn up to be valid, therefore the ample set  $\{1 \ll 3 | P_2\}$  is chosen nondeterministically to be the returned value. Thus there is only a single branch after the initial state when CPOR is applied. There are a total of 31, 14 and 5 states for LTS of process  $P$  in the situations where no POR, classic POR and CPOR are applied respectively.

#### 4.2.2 CPOR Algorithm

In this section, we will discuss the procedures for CPOR as given in Algorithm 4.1.  $CAmple$  returns an ample set which is a set of enabled configurations from the configuration  $c = (P, V)$ , and  $Visited$  is the stack of visited configurations. Each configuration  $c_a$  in the ample set, where  $c_a = (Proc, Val)$ , is associated with an action  $a_a = (Event, Time, EnableSiteType, EnableSiteId)$ , which caused the transition from  $c$  to  $c_a$ , that is  $c \xrightarrow{a_a} c_a$ . Henceforth, we use the dot-notation such as  $c_a.Proc$ ,  $c_a.Event$ , etc to denote the component values of  $c_a$  as well as the component values of its associated action  $a_a$ .  $P.Ample$  (line 2) is a set that stores ample set candidates that satisfy conditions A1 and A2, but yet to check for conditions A3 and A4. Procedure  $enableSubProcs(P)$  (line 3) returns the set of enabled child processes according to HCP graph of  $Orc$  expressions as shown in Figure 4.5, with an exception that for sequential



**Algorithm 4.1:** CAmple

---

```

1 procedure CAmple( $P, V, Visited$ )
2    $P.Amples := \emptyset$ ;
3   foreach  $sP \in enableSubProcs(P)$  do // A2' (2)
4     fillAmpleRec( $sP, V$ );
5     composeAmples( $P, sP, V$ );
6     foreach  $ample \in P.Amples$  do
7        $validAmple := true$ ;
8       foreach  $config \in ample$  do
9         if  $\neg config$  satisfies A3 // A3
10         $\vee config \in Visited$  // A4
11        then
12           $validAmple := false$ ;
13          break;
14        if  $validAmple$  then
15          return  $ample$ ;
16   return  $Enable((P, V))$ ;
17 procedure fillAmpleRec( $P, V$ )
18    $P.Amples := \{\{config : Enable((P, V))$ 
19      $| checkA2Local(config)\}\}$ ; // A2' (1)
20   if  $P$  is terminal process then
21     composeAmples( $P, P, V$ );
22   else
23     foreach  $sP \in enableSubProcs(P)$  do
24       fillAmpleRec( $sP, V$ );
25       composeAmples( $P, sP, V$ );
26 procedure composeAmples( $P, sP, V$ )
27    $P.Amples := P.Amples \cup reformAmples(sP.Amples, P)$ ;
28    $P.Amples := P.Amples \setminus \{\emptyset\}$ ; // A1
29 procedure checkA2Local( $config$ )
30   return( $config.EnableSiteType$  is stateless  $\vee$ 
31      $config.EnableSiteType$  is stateful  $\wedge$ 
32      $isPrivate(config.EnableSiteId)$ );

```

---

process  $P_s = A > x > B$ , it returns an empty set  $\{\}$  instead of  $\{A\}$ , and for pruning process  $P_p = A < x < B$ , it returns  $\{A\}$  instead of  $\{A, B\}$ . This exception is related to local criteria

of  $A2'(2)$ . Procedure  $fillAmpleRec(P, V)$  (line 17) retrieves the ample set candidates under valuation  $V$  and assigns it to  $P.Amples$ . In line 18,  $Enable(P, V)$  where  $c = (P, V)$  gives the set of all enabled configurations from the configuration  $c$ . Procedure  $checkA2Local(config)$  checks whether configuration  $config$  satisfies  $A2'(1)$ . Procedure  $isPrivate$  (line 32) checks whether the site with  $config.EnableSiteId$  as unique identity is *private* in  $Proc[P_G]$  where  $Proc$  is the process component of  $config$  and  $P_G$  is the argument  $P$  of procedure  $CAmple$  provided by user, which is the global process that has  $Proc$  as descendant process. The checking is done by syntax analysis. In  $Orc$ ,  $P$  is a terminal process (line 20) iff it is a site call. Procedure  $composeAmples(P, sP, V)$  (line 26) combines  $sP.Amples$  back into  $P.Amples$  under valuation  $V$ . Procedure  $reformAmples(sP.Amples, P)$  (line 27) restructures configurations within  $sP.Amples$  by operational semantics of  $Orc$ . For example, consider  $P = (1+x < x < 2)$ ,  $sP.Amples = \{\{c\}\}$ , and  $sP = 2$ . After making a transition,  $sP.Amples = \{\{c\}\}$ , where  $c$  is the configuration  $(stop, \emptyset)$  with  $c.Event = !2$ . After restructuring by  $reformAmples(sP.Amples, P)$ ,  $c$  becomes  $(1 + 2, \emptyset)$ , and  $c.Event = \tau$ , according to rule  $ASYM2V$  as stated below.

$$\frac{(2, \emptyset) \xrightarrow{!2} (stop, \emptyset)}{(1 + x < x < 2, \emptyset) \xrightarrow{\tau} (1 + 2, \emptyset)} \quad [ASYM2V]$$

When  $P = sP$ ,  $reformAmples(sP.Amples, P)$  will simply return  $sP.Amples$ . Subsequently, ample sets that are empty sets are filtered away (line 28). We continue on the discussion of procedure  $CAmple$ . To analyze whether an ample set  $ample$  is valid, the algorithm checks whether all configurations within satisfy conditions  $A3$  and  $A4$  (line 9, 10). If it turns out to be true, a valid ample set is found, and it will be returned immediately (line 14, 15). If no valid ample set has been found in line 3-15, all the enabled configurations from current configuration  $c = (P, V)$  will be returned (line 16). Regarding checking of condition  $A3$  (line 9), there are two kind of actions that might not be  $\emptyset$ -invisible, which are actions that contain publication events or actions that involved the update of global variables. Consider the metronome example, if we are checking property like whether  $!tick$  event can be executed

infinitely often, an action  $a$  with  $a.Event \neq !tick$  is not  $\emptyset$ -invisible. Another example is when we are checking whether  $tickNum < 0$  is true in all situations, where  $tickNum$  is a global variable, an action  $a$  with  $a : EnableSiteType = GUpdate$  is not  $\emptyset$ -invisible.

### 4.2.3 Soundness

**Lemma 4.2.3.** *Given any two actions  $a_1$  and  $a_2$  in the system, and let  $s_1$  and  $s_2$  be the enable sites of actions  $a_1$  and  $a_2$  respectively. If sites  $s_1$  and  $s_2$  are not descendant processes of the RHS process of some pruning combinators and datastores of sites  $s_1$  and  $s_2$  are disjoint, then action  $a_1$  is independent of action  $a_2$ .*

**Proof.** Actions  $a_1$  and  $a_2$  are dependent only when (a) action  $a_1$  could disable action  $a_2$  or vice versa or (b) starting from the same configuration, transitions  $a_1a_2$  and  $a_2a_1$  could result in different configurations. Situation (a) could happen if site  $s_1$  could possibly modify the datastore of site  $s_2$  or vice versa, or when sites  $s_1$  and  $s_2$  are the descendant processes of the RHS process of some pruning combinators. For the later case, consider  $x < x < (s_1|s_2)$ , if site  $s_1$  published a value, site  $s_2$  will be disabled immediately. Nevertheless, this case is ruled out by the assumption. Condition (b) could happen when sites  $s_1$  and  $s_2$  contain common datastore which they may modify and depend on. Therefore, conditions (a) and (b) are the results of having common datastore between sites  $s_1$  and  $s_2$ . This implies that if sites  $s_1$  and  $s_2$  have disjoint datastores, actions  $a_1$  and  $a_2$  are independent to each other.  $\square$   
**end.**

**Lemma 4.2.4.** *Given a configuration  $c = (P, V)$ , and process  $P_1$  as a descendant process of  $P$ . If  $P_1$  is not a descendant process of the LHS process of some sequential combinators, then a site  $S$  that is private in  $P_1[P]$ , is permanently private in  $P_1[c]$  as well.*

**Proof.** We prove by inspecting each rule in the operational semantics of *Orc* [93]. Only rule SEQ1V of operational semantics of *Orc* is possible to transfer the site reference from

a process  $p_1$  to another process  $p_2$ , while retaining process  $p_1$ . Consider HCPs under rule SEQ1V in Figure 4.4, a site  $S$  that is private in  $P_1[P_0]$  may not be private in  $P_1[P_2]$ , since  $P_3$  might have the access to the reference of site  $S$ . Therefore, if we exclude this situation by assuming  $P_1$  is not a descendant process of the LHS process of some sequential combinators, we prove the lemma.  $\square$  **end.**

Given a configuration  $c_g = (P_g, V_g)$ , and  $P_d$  as a descendant process of  $P_g$ , with associated configuration  $c_d = (P_d, V_d)$ .  $\mathbb{C}_{c_g}$  is defined as the set of configurations reachable by  $c_g$  in LTS;  $\mathbb{P}_{c_g}$  is defined as  $\{P|c = (P, V) \wedge c \in \mathbb{C}_{c_g}\}$ ;  $HCP(\mathbb{P}_{c_g})$  is defined as the HCPs for each global process in  $\mathbb{P}_{c_g}$ ;  $\mathbb{H}_{c_g}$  is defined as the union of processes within each HCP in  $HCP(\mathbb{P}_{c_g})$ ;  $\mathbb{H}_{c_g}[P_d]$  is the set of processes that contain process  $P_d$  and its corresponding descendant processes in respective HCPs in  $HCP(\mathbb{P}_{c_g})$ , and  $\mathbb{H}_{c_g}[P_d] \subseteq \mathbb{H}_{c_g}$ .

**Lemma 4.2.5.** *If an action  $a \in Act(c_d)$  satisfies A2' then the action is independent of any action  $b \in Act(c')$ , where  $c' = (P', V')$ , such that  $P' = \mathbb{H}_{c_g}/\mathbb{H}_{c_g}[P_d]$ , and  $V'$  is any valuation.*

**Proof.** Assume an action  $a \in Act(c_d)$  satisfies A2', and assume the action is dependent to an action  $b \in Act(c')$ . Let sites  $s_a$  and  $s_b$  be the enable sites of actions  $a$  and  $b$  respectively. By A2'(1), site  $s_a$  is a stateless site or stateful site that is private in  $p_a[P_g]$ . Site  $s_a$  could not be a stateless site since a stateless site does not have a datastore, and thus action  $a$  is trivially independent to any actions in the system by Lemma 1 and A2'(2). Therefore, site  $s_a$  is a stateful site that is private in  $p_a[P_g]$ . By Lemma 2 and A2'(2), site  $s_a$  is also permanently private in  $p_a[c_g]$ . By definition, datastores of site  $s_a$  and  $s_b$  are disjoint. By Lemma 1 and A2'(2), actions  $a$  and  $b$  are independent, a contradiction.  $\square$  **end.**

**Theorem 4.2.6.** *If any action  $a \in Act(c_d)$  satisfies A2', then  $AmpleAct(c_g) = Act(c_d)$  satisfies A2 for all traces in  $TS_{c_g}$ .*

**Proof.** Assume any action  $a \in Act(c_d)$  satisfies A2', and  $AmpleAct(c_g) = Act(c_d)$  does not satisfy A2 for some traces in  $TS_{c_g}$ . This means that there exists a finite execution fragment

$l = c \xrightarrow{a_1} c_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} c_n \xrightarrow{a_{n+1}} \dots$ , where actions  $a_1, \dots, a_n \notin Act(c_d)$  and action  $a_{n+1}$  depends on some actions in  $AmpleAct(c_g) = Act(c_d)$ . Since Lemma 3 holds, action  $a_{n+1}$  must be from  $PAct(c_d)/Act(c_d)$ , we denote the enable site of action  $a_{n+1}$  as  $S_{n+1}$ . Since site  $S_{n+1}$  is disabled initially in  $c_d$ , it means that it is enabled later by a site call from a process  $p' \in \mathbb{H}_{c_g}/\mathbb{H}_{c_g}[P_d]$ . For sites in process  $P_d$ , site calls from a process  $p' \in \mathbb{H}_{c_g}/\mathbb{H}_{c_g}[P_d]$  could only enable the sites that are shared in  $p_d[P'_g]$ , where  $P'_g$  is the global process of  $p'$ . We denote the set of datastores of the sites that are shared in  $p_d[P'_g]$  as  $\mathbb{D}_{share}$ , and datastore of  $S_{n+1}$  is in  $\mathbb{D}_{share}$ . On the other hand, by Lemma 2 and A2'(2), any action  $a \in Act(c_d)$  is enabled by a site that is permanently private in  $p_a[c_g]$ . By definition, datastore of the enable site of any action  $a \in Act(c_d)$  must not be found in  $\mathbb{D}_{share}$ . Therefore, action  $a_{n+1}$  is independent to all actions in  $Act(c_d)$  by Lemma 1 and A2'(2), a contradiction.  $\square$  **end.**

**Theorem 4.2.7.** *Algorithm CAmple is sound.*

**Proof.** To show the soundness of the algorithm, we need to show that the returned ample set satisfies conditions A1-A4. Checking of condition A1 is done at line 28. Conditions A3 and A4 are checked at the global process level (line 11, 12) at CAmple since they are only concerned with the property of global process configurations, i.e. their visibility and whether they have been visited before. By Theorem 1, satisfaction of condition A2' leads to satisfaction of condition A2. Condition A2'(1) is checked at line 19. Condition A2'(2) is guaranteed by constraining the procedure *enableSubProcs(P)* (line 3) not to return LHS process of a sequential process and RHS process of a pruning process.  $\square$  **end.**

### 4.3 Evaluation

Our approach has been realized in the ORC Module of Process Analysis Toolkit (PAT) [2, 86]. PAT is designed for systematic validation of distributed/concurrent systems using state-of-the-art model checking techniques. It can be considered as a framework for manufacturing

(A) Comparing difference POR methods

Model	Property	Size		States			Time(s)		
				CPOR	POR	No POR/CPOR	CPOR	POR	No POR/CPOR
Concurrent Quicksort	(1.1)	2	✓	58	1532	10594	0.08	1.13	5
		3	✓	69	3611	36794	0.11	8.48	74
		5	✓	237	-	-	0.68	-	-
Readers-Writers Problem	(2.1)	2	✗	106	1645	7620	0.07	1.12	4
		3	✗	152	18247	142540	0.11	14.86	101
		10	✗	472	-	-	0.49	-	-
Auction Management	(3.1)	N.A.	✓	869	-	-	0.6	-	-
	(3.2)	N.A.	✓	883	-	-	0.75	-	-

(B) Comparing Our Model Checker and Maude

Model	Property		States/Rewrites		Time(s)	
			Our	Maude	Our	Maude
Auction Management	(3.1)	✓	869	7052663	0.6	14.4
	(3.2)	✓	883	8613539	0.75	19.8

Table 4.1: Performance evaluation on model checking *Orc*'s model

model-checker. The data are obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 4GB RAM. ORC module supports verification of deadlock-freeness, Linear Temporal Logic (LTL) [82] property, and reachability property base on [68].

In part (A) of Table 4.1, three situations are compared: *CPOR* is the scenario where Compositional POR approach as described in Section 4.2 is applied; *POR* is the scenario where classic approach of POR that only considered the concurrency of processes at level 1 is applied; *No POR/CPOR* is the scenario where neither POR or CPOR is applied. Works such as [65, 20, 64] are not included for comparison. The reason is they optimized POR by restructuring or leveraging the information of processes at level 1, while CPOR is to extend POR to HCP, and this two approaches are orthogonal. This means that they could be used to optimized CPOR, in the same way they are used to optimized for POR, since CPOR and POR have same level 1 processes. In the table, ✓ and ✗ means the condition is satisfied and violated respectively. The results are omitted (shown as "-") for states and times, if it takes more than eight hours for verification. Model *Metronome* is described in Section 4.1. Property (1.1) is used to check for deadlock-freeness; property (1.2) is used to check whether

“tick” and “tock” are always possible to be published in the future. Property (1.3) is a reachability testing which is used to test whether it is possible to have two consecutive “tick”s or “tock”s published. Model *Concurrent Quicksort* is a variant of classic quicksort algorithm which emphasizes its concurrent perspective, as described in [61], for which the *size* column denotes the number of elements in the array to be sorted. Property (2.1) is used to verify whether elements in the array will eventually be sorted, and once sorted, it will remain sorted. Model *Readers-Writers Problem* is a famous computer science problem as described in [36], for which the *size* column denotes the number of readers. Property (3.1) verifies whether it is possible to reach a state that violates the mutual exclusion condition. Model *Auction Management* is the case study found in [9] which includes the use of external services. Please refer to [89] for the details of modeling of external services in our module. Property (4.1) is used to verify if an item has a bid on it, it is eventually sold; Property (4.2) is used to verify every item is sold to a unique winner. Part (B) of Table 4.1 is the result of comparing the effectiveness of our *Orc*’s model checker, with the work reported in [10, 11]. In [10, 11], Maude model checker is used for verification. Figures reported for number of rewrites and time usage for Maude model checker are from [11], which is run under 2.0GHz dual-core node with 4GB of memory. For all models, *CPOR* has shown state reduction over *POR*, and it also shows that our model checker is more efficient than the work reported in [10, 11].

## 4.4 Related work

This work is related to research on applying POR to hierarchical concurrent systems. Lang et al. [65], propose an approach of POR using compositional confluence detection. The proposed method works by analyzing the transitions of the individual process graph as well as the synchronization structure to identify the confluent transitions in the system

graph. Transitions within the individual process graphs (at level 1) are assumed to be dependent, thus all possible transitions will be generated for individual process graphs. While in our work, we further exploit the independency within each process recursively. Basten et al. [20], propose an approach to enhance POR via process clustering. The proposed method combines processes (at level 1) in clusters, and applies partial order reduction at proper cluster-level to achieve more reduction. The local transitions of each process (at level 1) are assumed to be dependent. Krimm et al. [64], propose an approach to compose the processes (at level 1) of an asynchronous communicating system incrementally, at the same time apply POR for the generated LTS. The local transitions of each process (at level 1) in the system are assumed to be dependent.

This work is related to research on verifying *Orc*. Yang et al. [39], propose an approach to translate the *Orc* language to Timed Automata for verification. In [87], we propose a preliminary idea in verifying *Orc* directly based on its semantics. In both approaches, no reduction to the LTS is considered. Alturki et al. [9, 10], propose an approach to translate the *Orc* language to rewriting logic for verification. An operational semantics of *Orc* in rewriting logic is defined, which is proved to be semantically equivalent to the operational semantics of *Orc*. To make the formal analysis more efficient, a reduction semantics of *Orc* in rewriting logic is again defined, which is proved to be semantically equivalent to the operational semantics of *Orc* in rewriting logic. We have compared the efficiency of our model checker with theirs in Section 4.3.





## Chapter 5

# Integrated Verification of Service Composition

Based on Service Oriented Architecture (SOA), Web services make use of open standards, such as WSDL [6] and SOAP [5], that enable the interaction among heterogeneous applications. A real-world business process may contain a set of services. A Web service is a single autonomous software system with its own thread of control. A fundamental goal of Web services is to have a collection of network-resident software services, so that it can be accessed by standardized protocols and integrated into applications or composed to form complex services which are called *composite services*. A composite service is constructed from a set of *component services*. Component services have their interfaces and functionalities defined based on their internal structures. While the technology for creating services and interconnecting them with a point-to-point basis has achieved a certain degree of maturity, there is a challenge to integrate multiple services for complex interactions. Web service composition standards have been proposed in order to address this challenge. The *de facto* standard for Web service composition is Web Services Business Process Execution Language (WS-BPEL) [56]. WS-BPEL is an XML-based orchestration business process lan-

guage. It provides basic activities such as service invocation, and compositional activities such as sequential and parallel composition to describe composition of Web services. BPEL is inevitably rich in concurrency and it is not a simple task for programmers to utilize concurrency as they have to deal with multi-threads and critical regions. It is reported that among the common bug types concurrency bugs are the most difficult to fix correctly, the statistic shows that 39% of concurrency bugs are fixed incorrectly [95]. Therefore, it is desirable to verify Web services with automated verification techniques, such as model checking [18].

There are two kinds of requirements of Web service composition, i.e., functional and non-functional requirements. Functional requirements focus on the functionalities of the Web service composition. Given a booking service, an example of functional requirement is that a flight ticket with price higher than \$2000 will never be purchased. The non-functional requirements are concerned with the Quality of Service (QoS). These requirements are often recorded in service-level agreements (SLAs), which is a contract specified between service providers and customers. Given a booking service, an example of non-functional requirements is that the service will respond to the user within 5 ms. Typical non-functional requirements include response time, availability, cost and so on. However, it is difficult for service designers to take the full consideration of both functional and non-functional requirements when writing BPEL programs.

Model checking is an automatic technique for verifying software systems [18], which helps find counterexamples based on the specification at the design time so that it could detect errors and increase the reliability of the system at the early stage. Currently, increasing number of complex service processes and concurrency are developed on Web service composition. Hence, model checking is a promising approach to solve this problem. Given functional and non-functional requirements, existing works [46, 49, 62, 72] only focus on verification of one aspect, and disregard the other, even though these two aspects are

inseparable. Different non-functional properties might have different aggregation functions for different compositional structures, and this poses a major challenge to integrate the non-functional properties into the functional verification framework.

In this work, we propose a method to verify BPEL programs against combined functional and non-functional requirements. A dedicated model checker is developed to support the verification. We make use of the labeled transition systems (LTSs) directly from the semantics of BPEL programs for functional verification. For non-functional properties, we propose different strategies to integrate different non-functional properties into the functional verification framework. We focus on three important non-functional properties in this work, i.e., availability, cost and response time. To verify availability and cost, we calculate them on-the-fly during the generation of LTS, and associate calculated values to each state in the LTS. Verification of response time requires an additional preprocessing stage, before the generation of LTS. In the preprocessing stage, response time tag is assigned to each activity that is participated in the service composition. With such integration, we are able to support combined functional and non-functional requirements.

The contributions of our work are summarized as follows.

1. We support integrated verification of functional and non-functional properties for Web service composition. To the best of our knowledge, we are the first work on such integration.
2. We capture the semantics of Web service composition using labeled transition systems (LTSs) and verify the Web service composition directly without building intermediate or abstract models before applying verification approaches, which makes our approach more suitable for general Web service composition verification.
3. Our approach has been implemented and evaluated on the real-world case studies, and this demonstrates the effectiveness of our method.

**Chapter Outline.** The rest of section is structured as follows. Section 5.1 describes the BPEL running example. Section 5.2 introduces QoS compositional model. Section 5.3 shows how to verify functional and non-functional properties. Section 5.4 provides the evaluation of our work. Section 5.5 reviews the related work.

## 5.1 Motivating Example

In our work, we assume that composite services are specified in the BPEL language. BPEL is the *de facto* standard for implementing composition of existing services by specifying an executable workflow using predefined activities. BPEL is an XML-based orchestration business process language for the specification of executable and abstract business processes. It supports control flow structures such as sequential and concurrency execution. In the following, we introduce the basic BPEL notations. `<receive>`, `<invoke>`, and `<reply>` are the basic communication activities which are defined to receive messages, execute component services and return messages respectively for communicating with component services. There are two kinds of `<invoke>` activities, i.e., synchronous and asynchronous invocation. Synchronous invocation activities are invoked and the process waits for the reply from the component service before moving on to the next activity. Asynchronous invocation activities are invoked and moving on to the next activity directly without waiting for the reply. The control flow of composite services is specified using the activities like `<sequence>`, `<while>`, `<if>` and `<flow>`. `<sequence>` is used to define the sequential ordering structure, `<while>` is used to define the loop structure, `<if>` is used to define the conditional choice structure, and `<flow>` is used to implement concurrency structure.

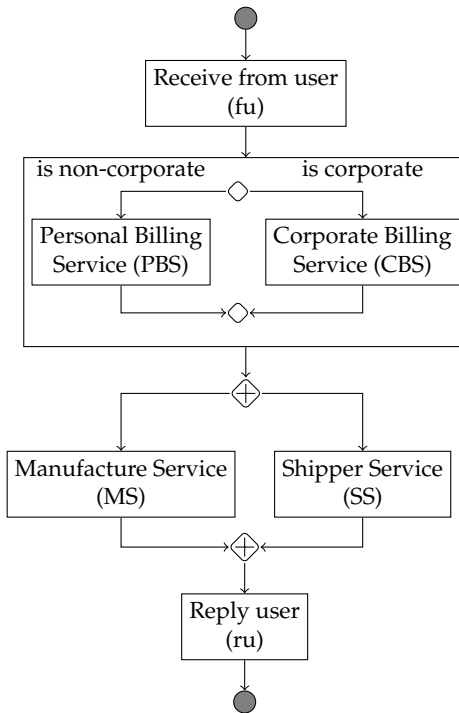


Figure 5.1: Computer Purchasing Service

### 5.1.1 Computer Purchasing Services (CPS)

In this section, we introduce the computer purchasing service (CPS), which is designed to allow users to purchase a computer online using credit cards. The workflow of CPS is illustrated in Figure 5.1.

CPS has four component Web services, namely Personal Billing Service (PBS), Corporate Billing Service (CBS), Manufacture Service (MS), Shipper Service (SS). CPS is initialized (denoted by ●) upon receiving the request from the customer ( $fu$ ) with the information of the customer and the computer that he wishes to purchase for. Subsequently, an `<if>` activity (denoted by ◇) is used for checking whether the customer is a corporate customer or non-corporate customer. If it is a corporate customer, CBS is invoked synchronously to bill the corporate customer, otherwise, PBS is invoked synchronously to bill the non-corporate customer with credit card information. Upon receiving the reply, a `<flow>`

activity (denoted by  $\diamond$ ) is triggered and MS and SS are invoked concurrently. MS is invoked synchronously to notify manufacture department for manufacturing the purchased computers. SS is invoked synchronously to schedule shipment for the purchased computers. Upon receiving the reply message from SS and MS, reply user (*ru*) is called to return the result of the computer purchasing to the customer. Then, the workflow of CPS has ended (denoted by  $\bullet$ ).

A property that CPS must fulfill is that it must invoke reply user (*ru*) within 5 *ms*. Notice that this property combines the functional (must invoke reply user (*ru*)) and non-functional (within 5 *ms*) requirements.

### 5.1.2 BPEL Notations

In order to present BPEL syntax compactly, we define a set of BPEL notations below:

- $rec(S)$  and  $reply(S)$  are used to denote “receive from” and “reply to” a service  $S$ ;
- $sInv(S)$  (resp.  $aInv(S)$ ) is used to denote synchronous (resp. asynchronous) invocation of a service  $S$ ;
- $P_1||P_2$  is used to denote <flow> activity, i.e., the concurrent execution of BPEL activities  $P_1$  and  $P_2$ ;
- $P_1 \triangleleft b \triangleright P_2$  is used to denote <if> activity, where  $b$  is a guard condition. Activity  $P_1$  is executed if  $b$  is evaluated true. Otherwise, activity  $P_2$  will be executed;
- $P_1;P_2$  is used to denote <sequence> activity, where  $P_1$  is executed followed by  $P_2$ .

We denote activities that contain other activities as *composite activities*, they are  $P_1||P_2$ ,  $P_1 \triangleleft b \triangleright P_2$  and  $P_1;P_2$ . For activities that do not contain any other activities, we denote them as *atomic activities*, they are  $rec(S)$ ,  $reply(S)$ ,  $sInv(S)$  and  $aInv(S)$ .

QoS Attribute	PBS	CBS	MS	SS
Response Time( <i>ms</i> )	1	2	3	1
Availability(%)	90	80	80	80
Cost(\$)	3	2	2	2

Table 5.1: QoS Attribute Values

## 5.2 QoS-Aware Compositional Model

In this section, we define the QoS compositional model used in this work and briefly introduce the semantics of BPEL, captured by labeled transition systems (LTSs). We introduce some definitions used in the semantic model in the following.

### 5.2.1 QoS Attributes

In this work, we deal with quantitative attributes that can be quantitatively measured using metrics. There are two classes of QoS Attributes, positive and negative attributes. Positive attributes (e.g., availability) have a good effect on the system, and therefore, they need to be maximized. Availability of the service is the probability of the service being available. Negative attributes (e.g., response time, cost) need to be minimized as they have the negative impact on the system. Response time of the service is defined as the delay between sending a request and receiving the response and cost of the service is defined as the money spent on the service. In this work, we assume the unit of response time, availability and cost to be millisecond (ms), percentage (%) and dollar (\$). Table 5.1 shows the information of response time, availability and cost of each component service for the CPS example as described in Section 5.1.1.

Given a component service  $s$  with  $n$  QoS attributes, we use a vector  $Q_s = \langle q_1(s), \dots,$



$q_n(s)$  to represent QoS attributes of the service  $s$ , where  $q_i(s)$  represents the value of  $i$ th attribute of the component service  $s$ . Similarly,  $Q'_{cs} = \langle q_1(cs)', \dots, q_n(cs)' \rangle$  is used to denote the QoS attributes of the composite service  $cs$ , where  $q_i(cs)'$  represents the  $i$ th attribute of the composite service  $cs$ .

### 5.2.2 QoS for Composite Services

A composite service  $S$  is constructed using a finite number of component services to reach a business goal. Let  $C = \langle s_1, s_2, \dots, s_n \rangle$  be the set of all component services that are used by  $S$ . The QoS of composite services is aggregated from the QoS of the component services, based on the service internal compositional structure, and the type of QoS attributes. Table 5.2 shows the aggregation functions for each compositional structure. We consider three types of QoS attributes: response time, availability and cost. For response time, in sequential composition, the response time of the composite service is aggregated by summing up the response time of each component service. As for parallel composition, the response time of the composite service is the maximum response time among that of each participating component service. For loop composition, the response time of the composite service is obtained by summing up the response time of the participating component service for  $k$  times, where  $k$  is the number of maximum iteration of the loop. And for conditional composition, the response time of the composite service is the maximum response time of  $n$  participating component services since it is not known that which guard is satisfied at the design phase. For availability, in sequential composition, the availability of the composite service is the product of that of all component services in the sequence because it means all component services are available during the sequential execution. It is similar to parallel and loop composition for aggregation of availability of the composite services. For conditional availability of the composite service, since one component service will be chosen at execution, therefore, we denote the availability as the minimum availability

QoS Attribute	Sequential	Parallel	Loop	Conditional
Response Time	$\sum_{i=1}^n q(s_i)$	$\max_{i=1}^n q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^n q(s_i)$
Availability	$\prod_{i=1}^n q(s_i)$	$\prod_{i=1}^n q(s_i)$	$(q(s_1))^k$	$\min_{i=1}^n q(s_i)$
Cost	$\sum_{i=1}^n q(s_i)$	$\sum_{i=1}^n q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^n q(s_i)$

Table 5.2: Aggregation Function

among all component services participated in the conditional composition. For cost, in sequential composition, the cost of the composite service is decided by the total cost of component services. For the conditional composition, the cost of the composite service is the maximum cost of  $n$  participating component services. Other common QoS attribute types can be aggregated in the similar way with these three attributes. For example, QoS attributes like reliability share the same aggregation function with availability.

### 5.2.3 Labeled Transition Systems

The QoS-aware composite model in this work is defined using labeled transition systems (LTS). In the following we define various terminologies that will be used in this work.

**Definition 5.2.1** (System State). *A system state  $s$  is a tuple  $(P, V, Q)$ , where  $P$  is the composite service process and  $V$  is a (partial) variable valuation that maps variables to their values,  $Q$  is a vector which represents QoS attributes of the composite service.*

Two states are equivalent iff they have the same process  $P$ , the same valuation  $V$  and the same QoS vectors  $Q$ . Given a system state  $s = (P, V, Q)$ ,  $Q = \langle r, a, c \rangle$  is a vector with three elements, where  $r, a, c \in \mathbb{R}_{\geq 0}$ , and  $0 \leq a \leq 1$ .  $r, a, c$  represent the response time, availability, and cost of the state  $s$ . The response time, availability, and cost are calculated from the

execution that starts at initial state  $s_0$  up to the state  $s$ . Henceforth, we use the notation  $Q(\text{ResponseTime})$ ,  $Q(\text{Availability})$  and  $Q(\text{Cost})$  to denote the value of  $r$ ,  $a$ , and  $c$  of QoS vector  $Q$ , respectively.

**Definition 5.2.2** (Composite Service Model). *A composite service model  $\mathcal{M}$  is a tuple  $(Var, P_0, V_0, F)$ , where  $Var$  is a finite set of variables,  $P_0$  is the composite service process, and  $V_0$  is an initial valuation that maps each variable to its initial value.  $F$  is a function which maps component services to their QoS attribute vectors.*

Given a composite service  $(Var, P_0, V_0, F)$ , an example of valuation  $V$  is  $\{var_1 \mapsto 1, var_2 \mapsto \perp\}$ , where  $var_1, var_2 \in Var$ , and  $var_2 \mapsto \perp$  is used to denote that  $var_2$  is undefined.

**Definition 5.2.3** (LTS). *An LTS is a tuple  $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$ , where*

- $S$  is a set of states,
- $s_0 \in S$  is the initial state,
- $\Sigma$  is a set of actions,
- $\rightarrow : S \times \Sigma \times S$  is a transition relation.

For convenience, we use  $s \xrightarrow{a} s'$  to denote  $(s, a, s') \in \rightarrow$  and we denote the LTS of a BPEL service  $\mathcal{M}$  as  $L(\mathcal{M})$ . Given a composite service model  $\mathcal{M} = (Var, P_0, V_0, F)$ ,  $L(\mathcal{M}) = (S, (P_0, V_0, Q_0), \Sigma, \rightarrow)$ .  $Q_0$  is the QoS attribute vector of the initial state, where the availability is 1, cost and response time are equal to 0. Give a state  $s \in S$ ,  $Enable(s)$  is denoted as the set of states reachable from  $s$  by one transition; formally,  $Enable(s) = \{s' | s' \in S \wedge a \in \Sigma \wedge s \xrightarrow{a} s' \in \rightarrow\}$ . An execution  $\pi$  of  $\mathcal{L}$  is a finite alternating sequence of states and actions  $\langle s_0, a_1, s_1, \dots, s_{n-1}, a_n, s_n \rangle$ , where  $\{s_0, \dots, s_n\} \in S$  and  $s_i \xrightarrow{a_{i+1}} s_{i+1}$  for all  $0 \leq i < n$ . We denote the execution  $\pi$  by  $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ . A state  $s$  is called reachable if there is an execution that ends in  $s$  and starts in an initial state.

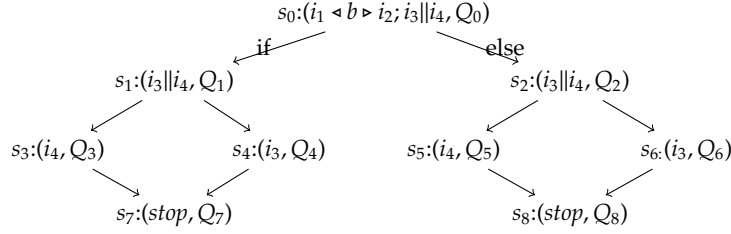


Figure 5.2: LTS of CPS where  $i_1$  is  $sInv(PBS)$ ,  $i_2$  is  $sInv(CBS)$ ,  $i_3$  is  $sInv(MS)$  and  $i_4$  is  $sInv(SS)$

Assume a composite service model is  $\mathcal{M} = (Var, P_0, V_0, F)$  and the LTS of  $\mathcal{M}$  is  $L(\mathcal{M}) = (S, s_0, \Sigma, \rightarrow)$ . Every action  $a \in \Sigma$  is triggered by an atomic activity. The atomic activities used in this work are  $rec(S)$ ,  $reply(S)$ ,  $sInv(S)$ , and  $aInv(S)$ , where  $S$  is the component service that the atomic activities are communicated with. For activities  $rec(S)$  and  $sInv(S)$ , they are required to wait for reply from component service  $S$  before continuing, therefore their availability, cost and response time are equivalent to the availability, cost and response time of component service  $S$ . For activities  $reply(S)$  and  $aInv(S)$ , they are not required to wait reply from the component service  $S$ , therefore they are regarded as internal operations. We assume the availability, cost and response time for an internal operations as 100%, \$0 and 0 ms respectively (see Section 5.3.3 for discussion). Given two states  $s = (P, V, Q)$ ,  $s' = (P', V', Q')$ , where  $s, s' \in S$ ,  $s \xrightarrow{a} s' \in \rightarrow$ , and  $a \in \Sigma$ , we use the function  $AtomAct(a)$  to denote the atomic activity that triggers the action  $a$ . As an example, given  $s = (sInv(S); rec(S), V, Q)$  and  $s' = (rec(S), V, Q)$ , the function  $AtomAct(a)$  returns the activity  $sInv(S)$ . We define the function  $ResponseTime(a)$ ,  $Availability(a)$  and  $Cost(a)$  to map the action  $a$  to the response time, availability, and cost of the activity returned by  $AtomAct(a)$ . Using the previous example,  $ResponseTime(a)$  is the response time of activity  $sInv(S)$ , which is essentially the response time of component service  $S$ .

The LTS of CPS as discussed in Section 5.1 is shown in Figure 5.2, where we omit the *Receive from user*( $fu$ ), *Reply user*( $ru$ ), all actions  $a \in \Sigma$ , and component  $V$  in the state for the reason of brevity. From state  $s_0$ , conditional activity  $i_1 \triangleleft b \triangleright i_2$  is enabled. Given that  $\{b \mapsto \perp\}$ ,

either  $i_1$  or  $i_2$  might be executed, therefore states  $s_1$  and  $s_2$  are evolved from state  $s_0$ . Noted that if guard  $b$  is defined, then only one branch is explored in the LTS. From state  $s_1$ , the flow activity  $i_3||i_4$  is enabled, and both activities  $i_3$  and  $i_4$  are allowed to execute. This leads to states  $s_3$  and  $s_4$ , respectively. State  $s_3$  evolves into state  $s_7$  after activity  $i_4$  is executed. *stop* activity in state  $s_7$  is a special activity which does nothing. Other states in LTS could be reasoned similarly. We assume that the upper bound on the number of iterations for loop activities is known, therefore, there is no recursive activities in BPEL.

### 5.3 Verification of Functional and Non-Functional Requirements

This section is devoted to discuss how to verify combined functional and non-functional requirements based on the LTS semantics of web service composition. Current works only verify one aspect of requirements, either functional or non-functional requirement, however, these two aspects are inseparable. For example, some property such as in the CPS example is required to reply the user within 5 ms, involves both functional and non-functional requirements. Therefore, we propose an approach to combine functional and non-functional requirements.

#### 5.3.1 Verification of Functional Requirement

To verify functional requirements of a BPEL program, LTS of the BPEL program is built from composite service model. We support the verification of deadlock-freeness, reachability of a state. To verify the LTL formulae, we make use of automata-based on-the-fly verification algorithm [35], by firstly translating a formula to a Büchi automaton and then checking emptiness of the product of the system and the automaton. For fairness checking, we utilize the on-the-fly parallel model checking based on Tarjan strongly connected components (SCC) detection algorithms similar to [84].

### 5.3.2 Integration of Non-Functional Requirement

In this section, we present our approach in integrating the non-functional requirements into verification framework. Different non-functional properties might have different aggregation functions for different compositional structures, and this poses a major challenge to integrate the non-functional properties into the functional verification framework. In the following, we adopt two different strategies in integrating the non-functional requirements. We first discuss our approach in integration of availability and cost, and following that, we discuss the integration of response time.

#### 5.3.2.1 Integration of Availability and Cost

In this section, we present our approach to integrate the availability and cost to the verification framework. Given two states  $s = (P, V, Q)$ ,  $s' = (P', V', Q')$ , where  $s, s' \in S$ ,  $s \xrightarrow{a} s' \in \rightarrow$ , and  $a \in \Sigma$ , the availability and cost of state  $s'$  is calculated using the following formulae:

$$\begin{cases} s'.Q(\text{availability}) = s.Q(\text{availability}) * \text{Availability}(a) \\ s'.Q(\text{cost}) = s.Q(\text{cost}) + \text{Cost}(a) \end{cases} \quad (5.1)$$

**Example.** We illustrate the integration using the LTS of CPS as shown in Figure 5.3. In state  $s_0$ , it has the initial availability of 1 and initial cost of \$0. From state  $s_0$ , it evolves into state  $s_1$  after invocation of  $i_1$ . Since  $i_1$  has availability of 0.9 and cost of \$3 (refer to Table 5.1), therefore the resulting QoS vector of state  $s_1$  is  $\langle r_1, 1 * 0.9, 0 + 3 \rangle = \langle r_1, 0.9, 3 \rangle$ . From state  $s_1$ , it evolves into state  $s_3$  after the invocation of  $i_3$ , and since  $i_3$  has availability of 0.8 and cost of \$2, the resulting QoS vector of state  $s_3$  is  $\langle r_3, 1 * 0.9 * 0.8, 0 + 3 + 2 \rangle = \langle r_3, 0.72, 5 \rangle$ . Other states are calculated similarly.

In general, given an execution  $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$  in  $L(\mathcal{M})$ , where  $\{s_0, \dots, s_n\} \in S$  and  $s_i \xrightarrow{a_{i+1}} s_{i+1} \in \rightarrow$ , for all  $0 \leq i < n$

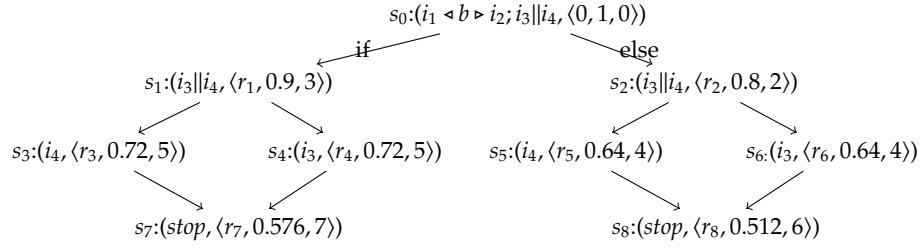


Figure 5.3: LTS of CPS with Availability and Cost, where  $i_1$  is sInv(PBS),  $i_2$  is sInv(CBS),  $i_3$  is sInv(MS) and  $i_4$  is sInv(SS)

$$\begin{cases} s_{i+1}.Q(\text{availability}) = s_0.Q(\text{availability}) * \prod_{m=1}^i \text{Availability}(a_m) \\ s_{i+1}.Q(\text{cost}) = s_0.Q(\text{cost}) + \sum_{m=1}^i \text{Cost}(a_m) \end{cases} \quad (5.2)$$

with  $s_0.Q = \langle 0, 1, 0 \rangle$ .

### 5.3.2.2 Integration of Response Time

One might naively think that we can adopt the method of calculating the cost as the method for calculating the response time. However, this would result in incorrect result. Refer to Figure 5.3, the value of response times  $r_2$ ,  $r_5$ ,  $r_6$ , and  $r_8$  will be 2 ms, 5 ms, 3 ms, and 6 ms respectively by using the method of calculating the cost in Section 5.3.2.1. In such case the value of  $r_8$  is incorrect. The reason is that it should be calculated as maximum of value of  $r_5$  and  $r_6$ , since parallelism allows both  $i_3$  and  $i_4$  to be executed simultaneously, and the total time for the response time is decided by the maximum response time of  $i_3$  and  $i_4$ . A challenge to evaluate the maximum time in state  $s_8$  is that the information of parallelism in state  $s_2$  ( $i_3 || i_4$ ) is removed in state  $s_5$  and state  $s_6$  (only left with  $i_3$  or  $i_4$ ). In order to retain this information, we preprocess the BPEL service model  $\mathcal{M}$  to associate with a time tag which will be used to calculate the response time in the LTS generation stage.

Algorithm 6.1 presents the main algorithm for preprocessing. Given a BPEL process  $P_0$ ,

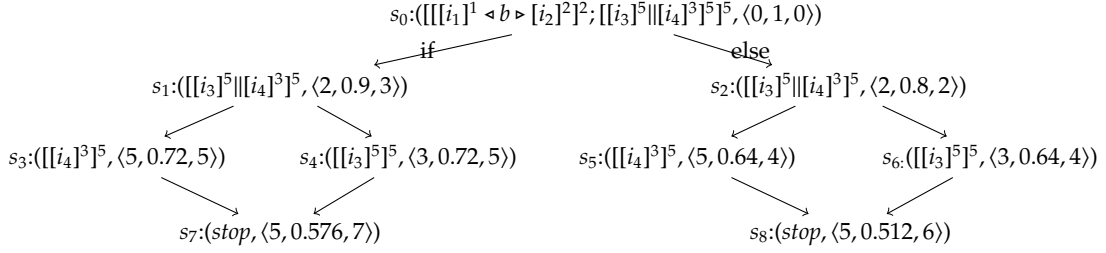


Figure 5.4: LTS of CPS with Response Time, Availability and Cost, where  $i_1$  is sInv(PBS),  $i_2$  is sInv(CBS),  $i_3$  is sInv(MS) and  $i_4$  is sInv(SS)

$TagTime(P_0, x)$  returns the process  $P'_0$  which is the process  $P_0$  with its internal activities associated with time tags. Given each activity  $Acv \in P_0$ , a value  $timetag \in \mathbb{R}_{\geq 0}$  is associated with  $Acv$ , denoted as  $Acv.timetag$ .  $Acv.timetag$  represents the total time delay from the start of process  $P_0$ , up to the completion of activity  $Acv$ . In the following, we describe the Algorithm 6.1. The function  $TagTime(P_0, x)$  is used to calculate the total time delay from the start of process  $P_0$  up to the completion of activity  $Acv$ . Variable  $x \in \mathbb{R}_{\geq 0}$  is the total time delay from the start of process  $P_0$  to the point just before the execution of activity  $Acv$ . Lines 1, 5, 9 and 11 are used to detect the structure of the activities. At line 1, if  $P$  is detected to be a sequential activity, activity  $A$  will be tagged with the delay  $x$  (line 2) as  $A$  is triggered once  $P$  is triggered. Subsequently, activity  $B$  will be tagged. Since activity  $B$  is executed after the completion of activity  $A$ , therefore the  $x$  is set to be the value of  $A.timetag$  (line 3). Finally, the  $timetag$  of  $P$  is the same as  $timetag$  of  $B$ , since the completion of activity  $B$  implies the completion of execution of process  $P$  (line 4). At line 5, if  $P$  is detected to be a concurrent or conditional activity, activity  $A$  and activity  $B$  will be tagged with value  $x$  (lines 6 and 7), since  $A$  and  $B$  are triggered at the same time once  $P$  is triggered. At line 8, the  $timetag$  of  $P$  is the maximum value of  $timetag$  of  $A$  and  $B$  (refer to Section 5.2.2 for details). If  $P$  is detected to be a synchronous receive activity or invocation activity, the  $timetag$  of  $P$  is set to the sum of  $x$  and  $ResponseTime(P)$  (line 10).

**Example.** In the following, we use an example to illustrate how to calculate the re-



---

**Algorithm 5.1:** Algorithm  $TagTime(P, x)$ 

---

**input** :  $P$ , the BPEL process  
**input** :  $x$ , the delay from the start to execution of process  $P$   
**output**:  $P'$ , process  $P$  with time tag

- 1 **if**  $P$  is  $A; B$  **then**
- 2      $TagTime(A, x)$ ;
- 3      $TagTime(B, A.timetag)$ ;
- 4      $P.timetag \leftarrow B.timetag$  ;
- 5 **else if**  $P$  is  $A\|B$  or  $A \triangleleft b \triangleright B$  **then**
- 6      $TagTime(A, x)$ ;
- 7      $TagTime(B, x)$ ;
- 8      $P.timetag \leftarrow \max(A.timetag, B.timetag)$  ;
- 9 **else if**  $P$  is  $rec(S)$  or  $sInv(S)$  **then**
- 10     $P.timetag \leftarrow x + ResponseTime(P)$ ;
- 11 **else if**  $P$  is  $reply(S)$  or  $aInv(S)$  **then**
- 12     $P.timetag \leftarrow x$ ;

---

sponse time for each state in the LTS. Given initial service process  $P_0 = sInv(PBS) \triangleleft b \triangleright sInv(CBS); (sInv(MS)\|sInv(SS))$ , we denote  $P'_0 = TagTime(P_0, 0)$  and

$$P'_0 = [ [ [sInv(PBS)]^1 \triangleleft b \triangleright [sInv(CBS)]^2 ]^2 ; [ [sInv(MS)]^5 \| [sInv(SS)]^3 ]^5 ]^5$$

where for each activity  $A \in P$ ,  $[A]^t$  is used to denote the activity  $A$  with  $A.timetag = t$ . Next, in the LTS generation stage, Algorithm 5.2 is used to calculate the response time for each state.

Given the process  $P$  of some state  $s \in S$ ,  $CalculateTime(P)$  in Algorithm 5.2 returns the total response time  $t \in \mathbb{R}_{\geq 0}$  from the initial state  $s_0$  to  $s'$ . The value  $t$  is assigned to  $Q(responseTime)$  for state  $s'$ . Lines 1, 6, 11 are used to detect the structure of the activities. We introduce a special activity *skip* to denote the completion of execution of an atomic activity. *skip* is used for the purpose of calculating the response time, and it will be removed after the calculation. At line 1, if  $P$  is detected to be a sequential activity, the activity  $A$  is then checked whether it is a *skip* activity. If it is (line 2), which implies that activity  $A$  has finished

---

**Algorithm 5.2:** Algorithm *CalculateTime(P)*

---

**input** :  $P$ , BPEL process with time tagged**output**:  $t \in \mathbb{R}_{\geq 0}$ , the time delay from the start of initial process  $P_0$  to the completion of  $P$ 

```

1 if  $P$  is  $A;B$  then
2   if  $A$  is skip then
3     return  $A.timetag$ ;
4   else
5     return  $CalculateTime(A)$ ;
6 else if  $P$  is  $A||B$  or  $A \triangleleft b \triangleright B$  then
7   if  $A$  is skip and  $B$  is skip then
8     return  $P.timetag$ ;
9   else
10    return  $CalculateTime(PreviousActive(P))$ ;
11 else if  $P$  is skip then
12   return  $P.timetag$ ;

```

---

execution,  $A.timetag$  is returned (line 3). Otherwise,  $CalculateTime(A)$  is invoked in order to determine the response time (line 5). At line 6, if  $P$  is detected to be a concurrent activity or conditional activity,  $A$  and  $B$  will be determined whether both are *skip* activities. If it is (line 7), which implies that  $P$  has finished execution,  $P.timetag$  is returned (line 8). Otherwise,  $CalculateTime(PreviousActive(P))$  is invoked in order to obtain the response time (line 10) where  $PreviousActive(P)$  is used to denote previous execution activity. For example, given  $s = (i_1||i_2, V, Q)$ ,  $s' = (skip||i_2, V', Q')$ , and  $s \xrightarrow{a} s' \in \rightarrow$ ,  $PreviousActive(skip||i_2)$  will return  $AtomAct(a) = i_1$ . At line 11,  $P$  is determined to be a *skip* activity implies that  $P$  has finished execution, therefore,  $P.timetag$  is returned (line 12). The value of  $timetag$  for each BPEL process is obtained using Algorithm 6.1.

**Example.** In Figure 5.4, given the initial state  $s_0$ , there are two branches due to the conditional process. If  $sInv(PBS)$  is executed, it will evolved into state  $s_1$  with process  $P'_1$  where

$$P'_1 = [ [ [ [skip]^1 ]^2 ; [ [sInv(MS)]^5 ] ] [ [sInv(SS)]^3 ]^5 ]^5$$

By running the Algorithm 5.2 for PBS to get the response time of PBS, it will return the value 2, therefore state  $s_1$  has the response time of 2 *ms*. After the calculating the response time, the *skip* are removed from  $P'_1$ , which result in process  $P_1 = [[sInv(MS)]^5][[sInv(SS)]^3]^5$  as shown in Figure 5.4. The calculation of other states is similar.

### 5.3.3 Discussion

If a system is verified that it does not satisfy the requirement that the response time is less than  $a$  ms in a state  $s$ , where  $a \in \mathbb{R}_{\geq 0}$ , it does not necessarily mean that such constraint will be violated in the state  $s$  during the execution. The response time is served as an estimated reference value. Furthermore, we do not take the response time, cost, and availability of internal operations into account. In reality, such information can be estimated using runtime monitoring method [71].

## 5.4 Evaluation

We evaluate our approach using three case studies. Each case study is a composite service represented as a BPEL process. The experiment data was obtained on a system using Intel Core I7 3520M CPU with 8GB RAM. The experimental results are summarized in Table 5.3.

### 5.4.1 Computer Purchasing Service (CPS)

As described in Section 5.1, CPS is used for allowing users to purchase a computer online using credit cards. The workflow of CPS is illustrated in Figure 5.1. The property *Reach* ( $replyUser \wedge (responseTime > 5)$ ) is to verify whether the activity *reply user* ( $ru$ ) can be reached with response time more than 5 *ms*. The result is invalid as shown in Table 5.3, which implies that if the *reply user* ( $ru$ ) is reached, it will be always be less than 5 *ms*, which is

Services	Property	Result	#State	#Transition	Time(s)
CPS	(replyUser $\wedge$ (responseTime>5))	invalid	21	29	0.0087
	$\square$ responseTime $\leq$ 5	valid	26	36	0.0089
	$\square$ availability>0.6	valid	26	36	0.0083
LS	Reach (replyUser $\wedge$ (responseTime>6))	invalid	106	241	0.0584
	$\square$ responseTime $\leq$ 6	valid	242	572	0.1866
TAS	Reach (replyUser $\wedge$ (responseTime>3))	invalid	128	287	0.0631
	$\square$ responseTime $\leq$ 3	valid	264	622	0.0642
	Reach (replyUser $\wedge$ (availability $\leq$ 0.3))	invalid	128	287	0.0437

Table 5.3: Experiment Results

the intended outcome we need. Properties  $\square$  *responseTime* $\leq$ 5 and  $\square$  *availability*>0.6 are LTL formulas, which are invariant properties denoted that the CPS's response time must always be less than two milliseconds and the CPS's availability is always larger than 50%. These two properties are both verified to be valid in the CPS system. The number of visited states, total transitions and time used for verification are listed in Table 5.3.

#### 5.4.2 Loan Service (LS)

The goal of a Loan Service (LS) is to provide users for applying loans. The loan approval system has several component systems, Loan Record Service (RS), Loan Approval Service (LAS), Customer Details Service (CDS), Customer Loan History Service (CLHS), Customer Credit Card History Service (CCHS), Customer Employment Information Service (CES) and Customer Property Information Service (CPIS). Upon receiving the request from a customer, CDS will be invoked synchronously. If the requested load amount is less than \$10000, CES is invoked and then RS is invoked to record the customer's loan information. After that, loan approval message will be replied to the customer. Otherwise, if the requested amount is not less than \$10000, CLHS, CCHS, CES and CPIS are invoked concurrently to obtain

more detailed information about the customer. Upon receiving all replies, LAS is invoked to determine whether to approve the load request of the customer or not. If the request is approved, RS is invoked synchronously and then loan approval message will be replied to the customer, otherwise, loan failure message will be replied to the customer. Two properties are verified for LS as listed in Table 5.3, we omit the discussion of the properties as they are similar to the properties of CPS.

### 5.4.3 Travel Agency Service (TAS)

Travel Agency Service (TAS) provides a service that helps users to arrange the flight, hotel, transport, etc., for a trip. Once the request is received from the user, Hotel Booking Service (HBS), Flight Booking Service (FBS), Local Transport Service (LoTS) and Local Agent Service (LAS) are triggered to search for available hotel, flight, local transportation and local travel agent concurrently that fulfill the user's requirements. If all four services have returned non-empty results, Record Booking Information Service (RBS) and Notify Agent Service (NAS) are invoked concurrently to store detailed booking information into the system and notify the agent about the customer's details. Finally, TAS replies the detailed booking information to the user. Otherwise, TAS replies booking failure result to the user. Three properties are verified for TAS as listed in Table 5.3. Properties  $Reach(replyUser \wedge (responseTime > 3))$  and  $\square responseTime \leq 3$  are similar to the properties verified in CPS, therefore we omit discussion of these two properties here. Property  $Reach(replyUser \wedge (availability \leq 0.3))$  is to verify whether *reply user (ru)* can be reached with the availability less than 0.3. The result is invalid as shown in Table 5.3, which implies that if the *reply user (ru)* is reached, the availability is always greater than 0.3, which is the intended result that we need.

The experiment shows that our approach can be used to verify the combined functional and non-functional property for real-world BPEL program efficiently.

## 5.5 Related Work

A number of approaches have been proposed to deal with requirements of Web service composition. These work can be divided into two major directions. One direction is to transform WS-BPEL processes into intermediate formal models specified in some formal languages and then verify the functional behaviors of the service composition based on the formal models. Foster et al. [46] translate BPEL processes into finite state processes notation. Qian et al. [77] transform BPEL processes into timed automata, and then use Uppaal as the model checker to verify the functional properties of the TA model, such as reachability. In [72, 67], the authors transform BPEL processes into Promela models and then use SPIN to verify the models. In [96], Yu et al. present a a lightweight specification language called PROPOLS to describe the temporal logic in a BPEL process. Different from these approaches, our current approach verifies functional properties of BPEL processes based on its semantics, thus it does not need to be translated into any other formal languages since there are some disadvantages of using intermediate models as mentioned at the beginning of this section. More important, our work combines verification of functional and non-functional requirements while works above only consider functional verification, which cannot verify functional and non-functional requirements at the same time.

Another direction has its focus on the non-functional aspect of BPEL processes. In [62], Koizumi and Koyama propose a performance model to estimate the processing execution time by integrating a Timed Petri Net model and statistical models. However, it only focuses on one type of non-functional requirements and does not consider the functional behaviors. In [49], Fung et al. propose a message tracking model to support QoS end-to-end management of BPEL processes. This work is based on the run-time data, which needs the deployment of the services, in addition, it does not consider the functional requirements of BPEL processes. Our approach verifies both functional and non-functional requirements at design time, which can detect errors at the early stage. In [94], Xiao et al. propose

a framework to use the simulation technique to verify the non-functional requirements before the service deployment, which is similar to our work. While their work only focus on non-functional aspect, our work supports verification of combined functional and non-functional properties. In [88], we propose a fully automatic approach for synthesis the local time requirement based on the given global time requirement of Web service composition. Different from them, our work focuses on checking LTL constraint satisfaction. And to the best of our knowledge, our work is the first one to verify combined functional and non-functional properties.

## Chapter 6

# Dynamic Synthesis of Response Time Requirement

Service-oriented architecture is a paradigm that promotes the building of software applications by using services as basic components. Services make their functionalities available through a set of operations accessible over a network infrastructure. To assemble a set of services to achieve a business goal, service composition such as BPEL (Business Process Execution Language) Orchestration [56] has been proposed. The service that is composed by service composition is called a *composite* service, and services that the composite service makes use of are called *component* services.

In business where timing is critical, a requirement on the service response time is often an important clause in service-level agreements (SLAs), which is the contractual basis between service consumers and service providers on the expected quality of service (QoS) level. Henceforth, we denote the response time requirement of composite services as *global time requirement*; and the set of constraints on the response times of the component services as the *local time requirement*. The response time of a composite service is highly dependent



on that of individual component services. It is therefore important to derive the local time requirement from the global time requirement so as to identify component services which could be used to build the composite service while satisfying the response time requirement.

Consider an example of a stock indices service, which has an SLA with the subscribed users, such that the stock indices would be returned in two seconds upon request. The stock indices service makes use of three component services, including a paid service, to request for the stock indices. The stock indices service provider would be interested to know the local time requirement of the component services.

BPEL is a service composition language that supports complex timing constructs and control flow structures such as concurrency. Such a combination of timing constructs, concurrent calls to external services, and complex control structures makes it a challenge to synthesize the local time requirement.

In this section, we present a fully automated technique for the synthesis of the local time requirement in BPEL. The approach works by performing dynamic analysis on the service composition, using techniques of parameter synthesis for real-time systems. For the synthesized local time requirement to be useful, it needs to be as weak as possible, to avoid discarding any service candidates that might be part of a feasible composition. This is particularly important, as often having a faster service would incur higher cost. To synthesize a better constraint that allows larger sets of feasible composition, we provide an extra analysis on the behavior of the composite service based on its associated labeled transition systems (LTSs). Our synthesis approach does not only avoid bad scenarios in the service composition, but also guarantees the fulfillment of global time requirement.

Our contributions are as follows:

1. Given a composite service, we develop a sound method for synthesizing the local

time requirement in the form of a set of constraints on the local service response times. The approach is implementation independent, therefore can be applied at the design stage of service composition.

2. We develop a fully automated tool to evaluate the proposed method and apply it to real-world case studies.

The synthesized local time requirement has multiple advantages. First, it allows the selection of feasible services from a large pool of services with similar functionalities but different local response times. Second, the designer can avoid over approximations on the local response times. An over approximation may lead the service provider to purchase a service at a higher cost, while a service at a lower cost with a slower response time may be sufficient to guarantee the global time requirement. Third, the local requirements serve as a safe guideline when component services are to be replaced or new services are to be introduced. Last but not least, the requirement synthesized by our method gives a quantitative measure of the *robustness* of the composite services. Indeed, if the global time requirement is satisfied given a local response time of 2 seconds, it may not be the case anymore for a value bigger than, but very close to, 2 seconds (e.g., 2.001 seconds). The constraint synthesized gives a measure of the upper bound until which each local response time can vary.

**Chapter Outline.** The rest of this section is structured as follows. Section 6.1 introduces a timed BPEL running example. Section 6.2 provides the necessary definitions and terminologies. Section 6.3 introduces our approach to dynamically analyze the BPEL process. Section 6.4 presents the synthesis algorithms and their soundness proofs. Section 6.5 discusses the application of our approach on two case studies. Section 6.6 reviews related works.

## 6.1 A Timed BPEL Example

BPEL [56] is an industrial standard for implementing composition of existing Web services by specifying an executable workflow using predefined activities. In this work, we assume the composite service is specified using the BPEL language. Basic BPEL activities that communicate with component Web services are `<receive>`, `<invoke>`, and `<reply>`, which are used to receive messages, execute component Web services and return values respectively. We denote them as communication activities. The control flow of the service is defined using structural activities such as `<flow>`, `<sequence>`, `<pick>`, `<if>`, etc. In this section, we illustrate a *Vehicle Booking Service* (VBS) as a running example in this work.

### 6.1.1 Vehicle Booking Service

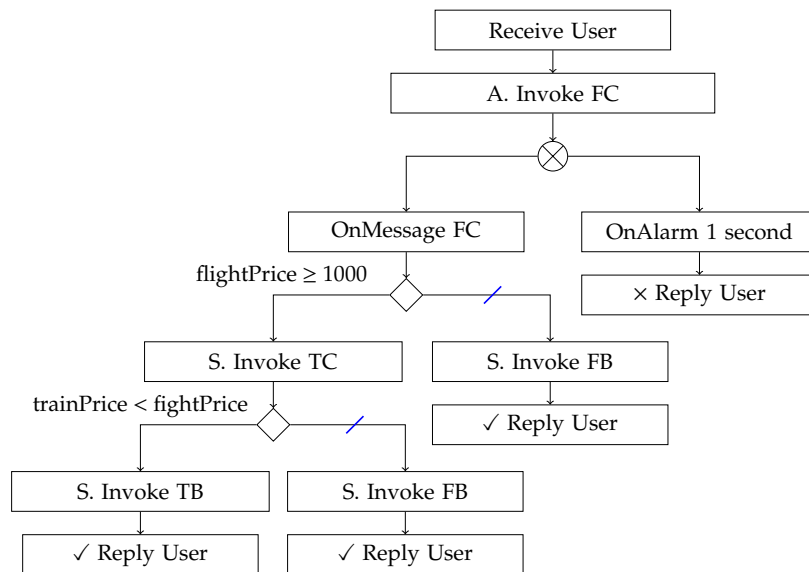


Figure 6.1: Vehicle Booking Service

The goal of Vehicle Booking Service (VBS) is to provide a combined flight and train booking services by integrating several independent existing services. It provides an SLA for its subscribed users, saying that it must respond within five seconds upon request.

The VBS has four component Web services: a flight checking service (FC), a train checking service (TC), a flight booking service (FB), and a train booking service (TB). The workflow of the VBS is sketched in Figure 6.1 in a tree structure. When a request is received from a subscribed customer (Receive User), it would asynchronously invoke (i.e., the system moves on after the invocation without waiting for the reply) the flight checking service (A. Invoke FC) to request for the flight information. A `<pick>` construct (denoted by  $\otimes$ ) is used here to wait incoming response (`<onMessage>`) from previous asynchronous invocation and timeout (`<onAlarm>`) if necessary. If the response from FC (`OnMessage FC`) is received within one second, an `<if>` branch (denoted by  $\diamond$ ) is used to check for the price of flight ticket. Otherwise, if the time-out occurs (`OnAlarm 1 second`), VBS stops waiting for the result from FC and notifies the user regarding the failure of getting flight information (Reply User). If the price of the flight ticket is less than 1000, the flight booking service (S. Invoke FB) is invoked synchronously (i.e., invoke and wait for reply) and then the booking result is replied to the user. Otherwise, TC is invoked synchronously (S. Invoke TC) to check the information of the train ticket followed by an `<if>` activity. If the price of the train ticket is less than that of flight, then TB is invoked synchronously (S. Invoke TB) to book the train ticket and the booking result is replied to the user (Reply User). Otherwise, the FB is invoked synchronously (S. Invoke FB) to book the flight ticket and the booking result is replied to the user (Reply User). The activity with a  $\checkmark$  (resp.  $\times$ ) represents the desired (resp. undesired) activity that ends the composition.

The global time requirement for VBS is that VBS should respond within five seconds upon request. It is of particular interest to know the local time requirements for services FC, TC, FB, and TB, so as to fulfill the global time requirement.

### 6.1.2 BPEL Notations

We succinctly recall the BPEL syntax notations below:

- $rec(S)$  and  $reply(S)$  are used to denote “receive from” and “reply to” a service  $S$ , respectively;
- $sInv(S)$  (resp.  $aInv(S)$ ) denotes the synchronous (resp. asynchronous) invocation of a service  $S$ ;
- $P||Q$  denotes the concurrent composition of BPEL activities  $P$  and  $Q$ ;
- $P;Q$  denotes the sequential composition of BPEL composition  $P$  and  $Q$ ;
- $P \triangleleft b \triangleright Q$  denotes the conditional composition, where  $b$  is a guard condition. If  $b$  is evaluated as true, BPEL activity  $P$  is executed, otherwise activity  $Q$  is executed;
- $pick(S \Rightarrow P, alrm(a) \Rightarrow Q)$  denotes the BPEL *pick* composition, which contains two branches of activities: *onMessage* activity and *onAlarm* activity, where either branch of the activity will be executed. *onMessage* activity is activated when the message from service  $S$  arrives within  $a$  seconds, where  $a \in \mathbb{R}_{>0}$ , and BPEL activity  $P$  is subsequently executed; *onAlarm* activity is activated at  $a$  seconds, and BPEL activity  $Q$  is subsequently executed. If the message arrives at exactly  $a$  seconds, then  $P$  or  $Q$  executes non-deterministically. Given a *pick* activity  $P$ , we use  $P.onMessage$  and  $P.onAlarm$  to denote the *onMessage* and *onAlarm* branches of  $P$  respectively.

## 6.2 Formal Model for Parametric Analysis

A composite service  $S$  makes use of a finite number of component services to accomplish a task. Let  $C = \{s_1, \dots, s_n\}$  be the set of all component services that are used by  $S$ . The response time of a service is reflected on the time spending on the communication activities. For example, assume that the only communication activity that communicates with component service  $s$  is  $sInv(s_1)$ . Upon invoking of service  $s$ , the construct  $sInv(s_1)$  waits for the reply. The response time of  $s_1$  is equivalent to the waiting time in  $sInv(s_1)$ . Therefore, by analyzing

the time spent in  $sInv(s_1)$ , we can get the response time of component service  $s_1$ . Given a composite service  $S$ , let  $t_i \in \mathbb{R}_{\geq 0}$  be the response time of component service  $s_i$  for  $i \in \{1, \dots, n\}$ , and let  $C_t = \{t_1, \dots, t_n\}$  be a set of component service response times that fulfill the global time requirement of service  $S$ . Because  $t_i$ , for  $i \in \{1, \dots, n\}$ , is a real number, there are infinitely many possible values, even in a bounded interval (and even if one restricts to rational numbers). A method to tackle this problem is to reason *parametrically*, by considering these response times as unknown constants, or *parameters*. Let  $u_i \in \mathbb{R}_{\geq 0}$  be the parametric response time of component service  $s_i$  for  $i \in \{1, \dots, n\}$ , and let  $C_u = \{u_1, \dots, u_n\}$  be the set of component service parametric response times. Using constraints on  $C_u$ , we can represent an infinite number of possible response times symbolically. The local time requirement of composite service  $S$  is specified as a constraint over  $C_u$ . An example of local time requirement is  $(u_1 \leq 6) \wedge (u_2 \leq 5)$ . This local time requirement specifies that, in order for  $S$  to satisfy the global time requirement, service  $s_1$  needs to respond within 6 time units, and service  $s_2$  needs to respond within 5 time units.

We review relevant definitions in the following.

### 6.2.1 Clocks, Parameters, and Constraints

The clocks, parameters and constraints that we use in this paper are similar to the ones used in the formalisms of timed automata [12] and parametric timed automata [14]. A *clock* is a variable with values in the set of non-negative real numbers  $\mathbb{R}_{\geq 0}$ . A clock is used to record the time passing of a communication activity. All clocks are progressing at the same rate.  $\mathcal{X}$  is defined as a universal set of clocks. Let  $X = \{x_1, \dots, x_H\} \subset \mathcal{X}$  (for some integer  $H$ ) be a finite set of clocks. A *clock valuation* is a function  $w : X \rightarrow \mathbb{R}_{\geq 0}$ , that assigns a non-negative real value to each clock.

A *parameter* is an unknown constant. Let  $\mathcal{U}$  denote the universal set of parameters, disjoint

with  $\mathcal{X}$ . Given a finite set of parameters  $U = \{u_1, \dots, u_M\} \subset \mathcal{U}$  (for some integer  $M$ ), a *parameter valuation* is a function  $\pi : U \rightarrow \mathbb{R}_{\geq 0}$  assigning a non-negative real value to each parameter. We can identify a valuation  $\pi$  with the point  $(\pi(u_1), \dots, \pi(u_M))$ . A *linear term* over  $X \cup U$  is an expression of the form  $\sum_{1 \leq i \leq N} \alpha_i z_i + d$  for some  $N \in \mathbb{N}$ , with  $z_i \in X \cup U$ ,  $\alpha_i \in \mathbb{R}_{\geq 0}$  for  $1 \leq i \leq N$ , and  $d \in \mathbb{R}_{\geq 0}$ . Given  $X \subset \mathcal{X}$  and  $U \subset \mathcal{U}$ , an *inequality* over  $X$  and  $U$  is  $e < e'$  with  $< \in \{<, \leq\}$ , where  $e$  and  $e'$  are linear terms over  $X \cup U$ . A *constraint* is a conjunction of inequalities. We denote by  $C_{X \cup U}$  the set of all constraints over  $X$  and  $U$ . Henceforth, we use  $w$  (resp.  $\pi$ ) to denote a clock (resp. parameter) valuation.

Let  $C \in C_{X \cup U}$ ,  $C[\pi]$  denote the constraint over  $X$  obtained by replacing each  $u \in U$  with  $\pi(u)$  in  $C$ . Similarly,  $C[\pi][w]$  denotes the constraint obtained by replacing each clock  $x$  in  $C[\pi]$  with  $w(x)$ . We write  $(w, \pi) \models C$ , if  $C[\pi][w]$  evaluates to true.  $C$  is *empty*, if there does not exist a parameter valuation  $\pi$ , such that  $\pi \models C$ ; otherwise  $C$  is *non-empty*. We define  $C^\dagger = \{x + d \mid x \in C \wedge d \in \mathbb{R}_{\geq 0}\}$ , as *time elapsing* of  $C$ , i.e., the constraint over  $X$  and  $U$  obtained from  $C$  by delaying an arbitrary amount of time. Given two constraints  $C_1, C_2 \in C_{X \cup U}$ ,  $C_1$  is *included* in  $C_2$ , denoted by  $C_1 \subseteq C_2$ , if  $\forall w, \pi : (w, \pi) \models C_1 \Rightarrow (w, \pi) \models C_2$ . Similarly,  $C_1$  is *strictly included* in  $C_2$ , denoted by  $C_1 \subset C_2$ , iff  $C_1 \subseteq C_2$  and  $C_1 \neq C_2$ .

## 6.2.2 Syntax of Composite Services

**Definition 6.2.1** (Composite Service Model). A composite service model  $\mathcal{M}$  is a tuple  $(Var, V_0, P_0)$ , where  $Var$  is a finite set of variables,  $V_0$  is an initial valuation that maps each variable to its initial value, and  $P_0$  is the composite service process.

Parametric composite service models extend composite service models with *parameters* in place of constants.

**Definition 6.2.2** (Parametric Composite Service Model). A parametric composite service model  $\mathcal{M}$  is a tuple  $(Var, V_0, U, P_0, C_0)$ , where  $Var$  is a finite set of variables,  $V_0$  is an initial

valuation that maps each variable to its initial value,  $U$  is a finite set of parameters,  $P_0$  is the composite service process, and  $C_0$  is the initial constraint.

Given a service model  $\mathcal{M}$  with a parameter set  $U = \{u_1, \dots, u_m\}$ , and given a parameter valuation  $(\pi(u_1), \dots, \pi(u_m))$ ,  $\mathcal{M}[\pi]$  denotes the valuation of  $\mathcal{M}$  with  $\pi$ , viz., the model  $(Var, V_0, U, P_0, C)$ , where  $C$  is  $C_0 \wedge \bigwedge_{i=1}^M (u_i = \pi(u_i))$ . Note that  $\mathcal{M}[\pi]$  can be seen as a non-parametric service model  $(Var, V_0, P_0[\pi])$ , where  $P_0[\pi]$  corresponds to  $P_0$  where each occurrence of a parameter  $u_i$  has been replaced with its valuation  $\pi(u_i)$ .

### 6.2.3 Semantic Models

In this work, we capture the semantics of composite services using labeled transition systems (LTSs). The behavior of composite service is affected by the input data. Since the input data is unpredictable, in order to reason about the general behavior of composite service, it is useful to obtain a local time requirement which could guarantee global time requirement for any input data. For example, given a conditional expression  $P \triangleleft b \triangleright Q$ , the execution of activity  $P$  or activity  $Q$  is based on the valuation of  $b$ ; but if we choose to abstract from data, either  $P$  or  $Q$  will be executed non-deterministically. A state of the LTS represents a status of the composite service. Informally, a concrete state is a state that contains data information, and the LTS that contains concrete states is denoted as a concrete LTS. An abstract state is a state which abstracts away data information, and the LTS that contains abstract states is denoted as an abstract LTS. Our dynamic analysis is based on the abstract LTS. In the following, we provide the formal definition of various terminologies that are used in this work.

**Definition 6.2.3** (Concrete State). A concrete state  $s_c \in S_c$  is a tuple  $(V, P, C, D)$ , where  $V$  is a valuation of the variables (i.e., a function that maps a variable name to its value),  $P$  is a composite service process,  $C$  is a constraint over  $C_X$ , and  $D$  is the (real-valued) duration from the initial state



$s_0$  to the beginning of the state  $s$ .

**Definition 6.2.4** (Concrete Labeled Transition System). A concrete labeled transition system is a tuple  $\mathcal{L}_c = (S_c, s_0^c, \Sigma, \rightarrow_c)$ , where

- $S_c$  is a set of concrete states,
- $s_0^c \in S_c$  is the initial state,
- $\Sigma$  is the universal set of actions,
- $\rightarrow_c : S_c \times \Sigma \times S_c$  is a transition relation.

**Definition 6.2.5** (Abstract State). An abstract state  $s \in S$  is a tuple  $(P, C, D)$ , where  $P$  is a composite service process,  $C$  is a constraint over  $C_{X \cup U}$ , and  $D$  is the (parametric) duration from the initial state  $s_0$  to the beginning of the state  $s$  (i.e., a linear term on the parameters).

**Definition 6.2.6** (Abstract Labeled Transition System). An abstract labeled transition system is a tuple  $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$ , where

- $S$  is a set of abstract states,
- $s_0 \in S$  is the initial state,
- $\Sigma$  is the universal set of actions,
- $\rightarrow : S \times \Sigma \times S$  is a transition relation.

Henceforth, when clear from the context, we refer to an abstract state (resp. abstract LTS) as a state (resp. LTS).

The following definitions could be easily extended to concrete states and concrete LTSs. Given an LTS  $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$ , a state  $s \in S$  is said to be a *terminal state* if there does not exist a state  $s' \in S$  and an action  $a \in \Sigma$  such that  $(s, a, s') \in \rightarrow$ ; otherwise,  $s$  is said to be

a *non-terminal state*. There is a *run* from a state  $s$  to state  $s'$ , where  $s, s' \in S$ , if there exist a set of states  $\{s_1, \dots, s_n\} \subseteq S$  and a set of actions  $\{a_1, \dots, a_n\} \subseteq \Sigma$  such that  $s_1 = s$ ,  $s_n = s'$ , and  $\forall i \in \{1, \dots, n-1\}, (s_i, a_i, s_{i+1}) \in \rightarrow$ . A *complete run* is a run that starts in the initial state  $s_0$  and ends in a terminal state. Given a state  $s \in S$ , we use  $Enable(s)$  to denote the set of states reachable from  $s$ ; formally,  $Enable(s) = \{s' | s' \in S \wedge a \in \Sigma \wedge (s, a, s') \in \rightarrow\}$ . Given a state  $s = (P, C, D)$ , we use the notation  $s.P$  to denote the component  $P$  of  $s$ , and similarly for  $s.C$  and  $s.D$ . These actions are used to update the active state  $s_a \in S$ , and store them as part of the execution run  $\Pi$ .

Given  $\mathcal{M} = (Var, U, P_0, C_0)$ , the *global time requirement* for  $\mathcal{M}$  requires that, for every state  $(P, C, D)$  reachable from the initial state  $(P_0, C_0, 0)$  in the LTS, the constraint  $D \leq T_G$  is satisfied, where  $T_G \in \mathbb{R}_{\geq 0}$  is the *global time constraint*. The *local time requirement* requires that if the response times of all component services of  $\mathcal{M}$  satisfy the *local time constraint*  $C_L \in C_U$ , then the service  $S$  satisfies the global time requirement.

### 6.3 Dynamic Analysis with LTS

Our approach for synthesizing local time constraint for component services is based on the dynamic analysis of the constraint of each state in the LTS of the composite service  $\mathcal{M}$ . In this section, we present how we analyze the LTS with real-time semantics using parametric techniques.

In order to analyze the LTS with real-time semantics, we use *clocks* to record the elapsing of time. The clock formalism has been used to record the time elapsing in several formalisms, in particular in timed automata (TA) [12]. In TAs, the clocks are defined as part of the models and state space. It is known that the state space of the system could grow exponentially with the number of clocks and that the fewer clocks, the more efficient real-time model checking is [21]. An alternative approach is to create clocks on the fly when necessary,

$Act(A(S), x)$	$= A(S)_x$	A1
$Act(mpick, x)$	$= mpick_x$	A2
$Act(A(S)_{x'}, x)$	$= A(S)_{x'}$	A3
$Act(mpick_{x'}, x)$	$= mpick_{x'}$	A4
$Act(P \oplus Q, x)$	$= Act(P, x) \oplus Act(Q, x)$	A5
$Act(P; Q, x)$	$= Act(P, x); Q$	A6

where  $A \in \{rec, sInv, aInv, reply\}$ ,  $\oplus \in \{\|\|, \langle b \rangle\}$ ,  
and  $mpick = pick(S \Rightarrow P_t, alrm(a) \Rightarrow P_a)$ .

Figure 6.2: Activation function

and have them pruned when no longer needed. This approach was initially proposed for (parametric) stateful timed CSP [83, 15]. This allows smaller state space compared to the explicit clock approach; we refer to this second approach as the *implicit clock approach*. We use here the implicit clock approach when analyzing the BPEL model with LTSs.

### 6.3.1 Clock Activation

Clocks are implicitly associated with timed processes. For instance, given an action  $sInv(s)$ , a clock starts ticking once the process becomes activated. To introduce clocks on the fly, we define an activation function  $Act$  in Fig. 6.2. Given a process  $P$ , we denote by  $P_x$  the corresponding process that has been associated with clock  $x$ . The activation function will be called when a new state  $s$  is reached to assign a new clock for each newly activated communication activity. Rules A1 and A2 state that a new clock is associated with BPEL communication activity if it is newly activated. Rules A3 and A4 state that if a BPEL communication activity has already been assigned a clock, it will be never reassigned. Rules A5 and A6 state that function  $Act$  is applied recursively for activated child activities for BPEL structural activities.

$$\begin{aligned}
idle(A(S)_x) &= x \leq t_S & \text{I1} \\
idle(B(S)_x) &= (x = 0) & \text{I2} \\
idle(P \oplus Q) &= idle(P) \wedge idle(Q) & \text{I3} \\
idle(P; Q) &= idle(P) & \text{I4} \\
idle(mpick_x) &= x \leq t_S \wedge x \leq a & \text{I5}
\end{aligned}$$

where  $A \in \{rec, sInv\}$ ,  $B \in \{aInv, reply\}$ ,  $\oplus \in \{\parallel, \langle b \rangle\}$ ,  
 $mpick = pick(S \Rightarrow P_t, alrm(a) \Rightarrow P_a)$ , and  $t_S$  is the  
parametric response time of service  $S$ .

Figure 6.3: Idling function

### 6.3.2 Idling Function

We define in Fig. 6.3 the function *idle* that, given a state  $s$ , calculates how long an activity  $A$  can idle in  $s$ . The result is a constraint over the clocks and the parameters. Rule I1 considers the situation that the communication requires to wait for the response of component services  $S$ , and the value of clock  $x$  must not be larger than the response time parameter  $t_S$  of the service. Rule I2 considers that the situation that no waiting is required. Rules I3 to I5 state consider that the function *idle* is applied recursively for activated child activities of a BPEL structural activity.

### 6.3.3 Bad Activity

Given a BPEL service  $\mathcal{M}$ , we define a *bad activity* as an activity such that its execution will immediately cause the system to violate the global time requirement. The bad activity is executed due to the failure of receiving the responses from component services. In the case of the VBS example, it corresponds to a situation where the component service TC fails to response within one second. To distinguish the bad activities, we allow the user to annotate a BPEL activity  $A$  as a bad activity, denoted by  $[A]_{bad}$ . The annotation can be achieved, for example, by using extension attribute of BPEL activities. The execution of activity  $[A]_{bad}$  will result the LTS of  $\mathcal{M}$  to end in an undesired terminal state, which we denote as a *bad*

*state*. A terminal state which is not a bad state is called a *good state*. The synthesized local time requirement needs to guarantee the avoidance of all bad states and the reachability of at least one good state.

### 6.3.4 Operational Semantics

We define the operational semantics of a parametric composite service model as a set of firing rules. A state  $s = (P, C, D)$  that satisfies a rule could evolve to another state  $s' = (P', C', D')$  according to the rule. We list the rules in Fig. 6.4.

$$\begin{array}{c}
\frac{x = t_s \wedge C^\uparrow}{(sInv(S)_x, C, D) \xrightarrow{e} (Stop, x = t_s \wedge C^\uparrow, D + t_s)} [rSInv] \quad \frac{}{(A \triangleleft b \triangleright B, C, D) \xrightarrow{e} (A, C, D)} [rCond1] \\
\frac{x = t_s \wedge C^\uparrow}{(rec(S)_x, C, D) \xrightarrow{e} (Stop, x = t_s \wedge C^\uparrow, D + t_s)} [rRec] \quad \frac{}{(A \triangleleft b \triangleright B, C, D) \xrightarrow{e} (B, C, D)} [rCond2] \\
\frac{x = 0 \wedge C^\uparrow}{(reply(S)_x, C, D) \xrightarrow{e} (Stop, x = 0 \wedge C^\uparrow, D)} [rReply] \quad \frac{(A, C, D) \xrightarrow{e} (A', C', D'), A' \neq Stop}{(A \sharp B, C, D) \xrightarrow{e} (A' \sharp B, C', D')} [rSeq1] \\
\frac{x = 0 \wedge C^\uparrow}{(aInv(S)_x, C, D) \xrightarrow{e} (Stop, x = 0 \wedge C^\uparrow, D)} [rAInv] \quad \frac{}{(A \sharp B, C, D) \xrightarrow{e} (Stop, C', D'), C \wedge C'} [rSeq2] \\
\frac{}{(A, C, D) \xrightarrow{e} (A', C', D'), C' \wedge idle(B)} [rFlow1] \quad \frac{(x = t_s) \wedge idle(mpick_x) \wedge C^\uparrow}{(mpick_x, C, D) \xrightarrow{e} (P_t, (x = t_s) \wedge idle(mpick_x) \wedge C^\uparrow, D + t_s)} [rPick1] \\
\frac{}{(B, C, D) \xrightarrow{e} (B', C', D'), C' \wedge idle(A)} [rFlow2] \quad \frac{(x = a) \wedge idle(mpick_x) \wedge C^\uparrow}{(mpick_x, C, D) \xrightarrow{e} (P_a, (x = a) \wedge idle(mpick_x) \wedge C^\uparrow, D + a)} [rPick2] \\
\frac{}{(A \parallel B, C, D) \xrightarrow{e} (A' \parallel B, C' \wedge idle(B), D')} \\
\frac{}{(A \parallel B, C, D) \xrightarrow{e} (A \parallel B', C' \wedge idle(A), D')}
\end{array}$$

Figure 6.4: Operational semantics

As an example, for rule  $rSInv$ , given state  $s = (sInv(S)_x, C, D)$ , it could evolve into state  $s' = (Stop, x = t_s \wedge C^\uparrow, D + t_s)$  via action  $e \in \Sigma$  if the precondition  $x = t_s \wedge C^\uparrow$  is satisfied, where  $Stop$  is the activity that does nothing. Other firing rules can be described similarly.

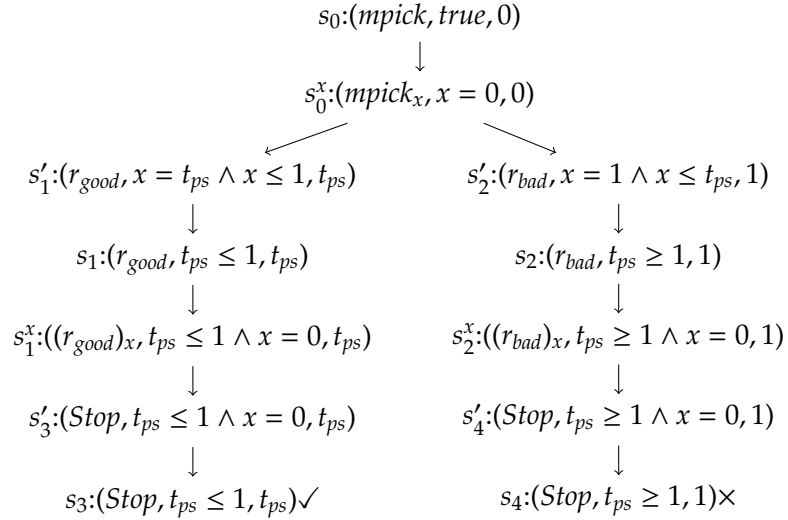
### 6.3.5 State Space Exploration

Let  $Y = \langle x_0, x_1, \dots \rangle$  be a sequence of clocks. Starting from the initial state  $s_0 = (P_0, C_0, 0)$ , we iteratively construct successor states as follows. Given a state  $(P, C, D)$ , a clock  $x$  which is not currently associated with  $P$  is picked from  $Y$ . The state  $(P, C, D)$  is transformed into  $(Act(P, x), C \wedge x = 0, D)$ , i.e., timed processes which just become activated are associated with  $x$  and  $C$  is conjuncted with  $x = 0$ . Then, a firing rule is applied to get a target state  $(P', C', D')$ . Lastly, clocks which do not appear within  $P'$  are pruned from  $C'$ . Observe that one clock is introduced and zero or more clocks may be pruned during a transition. Actually, a clock is introduced only if necessary; if the activation function does not activate any subprocess, this new clock is not created.

### 6.3.6 Application to an Example

Consider a composite service  $\mathcal{M}$ , where the LTS of  $\mathcal{M}$  is shown in Figure 6.5.

- At state  $s_0$ , activation function assigns clock  $x$  to record time elapsing of pick activity  $mpick$ , with  $x$  initialized to zero time unit. The tuple becomes the intermediate state  $s_0^x = (mpick_x, x = 0, 0)$ .
- From intermediate state  $s_0^x$ , it could evolve into the intermediate state  $s_1'$  by applying the rule  $rPick1$ , if the constraint  $c_1 = (true \wedge x = t_{PS} \wedge idle(mpick_x) \wedge (x = 0)^\uparrow)$ , where  $idle(mpick_x) = (x \leq t_{PS} \wedge x \leq 1)$  and  $(x = 0)^\uparrow = x \geq 0$ , is satisfiable. Intuitively,  $c_1$  denotes the constraint where  $t_{PS}$  time units elapsed since clock  $x$  has started. In fact,  $c_1$  is satisfiable (for example with  $t_{PS} = 0.5$  and  $x = 0.5$ ). Therefore, it could evolve into the intermediate state  $s_1' = (r_{good}, x = t_{PS} \wedge idle(mpick_x) \wedge (x = 0)^\uparrow, t_{PS}) = (r_{good}, x = t_{PS} \wedge x \leq 1, t_{PS})$ . Since clock  $x$  is not used anymore in  $s_1'.P$  which is  $r_{good}$ , it is eliminated using variable elimination techniques such as Fourier-Motzkin [80]. After



where  $mpick = pick(PS \Rightarrow r_{good}, alm(1) \Rightarrow r_{bad})$ ,  $r_{good} = reply(User)$ ,  $r_{bad} = [reply(User)]_{bad}$ ,  $t_{PS}$  is the parametric response time of service  $PS$ .

Figure 6.5: LTS of service  $\mathcal{M}$

elimination of clock variable  $x$  and simplification of the expression, the intermediate state  $s_1'$  becomes the state  $s_1 = (r_{good}, t_{PS} \leq 1, t_{PS})$ .

- From intermediate state  $s_0^x$ , it could also evolve into the intermediate state  $s_2'$ , by applying the rule  $rPick2$ , if the constraint  $c_2 = (true \wedge x = 1 \wedge idle(mpick_x) \wedge (x = 0)^\uparrow)$ , where  $idle(mpick_x) = (x \leq t_{PS} \wedge x \leq 1)$  and  $(x = 0)^\uparrow = x \geq 0$ , is satisfiable. It is easy to see that  $c_2$  is satisfiable; therefore, it could evolve into the intermediate state  $s_2' = (r_{bad}, x = 1 \wedge x \leq t_{PS}, 1)$ . After clock pruning from intermediate state  $s_2'$ , it becomes state  $s_2 = (r_{bad}, t_{PS} \geq 1, 1)$ .
- From state  $s_1$ , activation function assigns clock  $x$  for reply activity  $r_{good}$ , and it evolves into intermediate state  $s_1^x$ . From intermediate state  $s_1^x$ , it could evolve into intermediate state  $s_3'$  by applying rule  $rReply$ , if the constraint  $c_3 = (t_{PS} \leq 1 \wedge x = 0 \wedge (x = 0)^\uparrow)$  is satisfiable. In fact it is, and therefore it evolves into state  $s_3' = (stop, t_{PS} \leq 1 \wedge x = 0, t_{PS})$ . After pruning of the used clock, it evolves into the terminal state  $s_3 = (stop, t_{PS} \leq$

$1, t_{PS}$ ). Since the terminal state is not caused by a bad activity,  $s_3$  is considered as a good state, denoted by  $\checkmark$  in Figure 6.5.

- From state  $s_2$ , it could also evolve into terminal state  $s_4 = (stop, t_{PS} \geq 1, 1)$  with similar reason as above. Since the terminal state is caused by a bad activity, it is considered as a bad state, denoted by  $\times$  in Figure 6.5.

Note that all intermediate states  $s'_i$  and  $s^x_i$ , where  $i \in \mathbb{N}$  and  $0 \leq i \leq 4$ , are served as illustrative purpose, and are not part of the state space for composite service  $\mathcal{M}$ . Henceforth, the intermediate states will be omitted for the sake of conciseness.

## 6.4 Local Time Requirement Synthesis

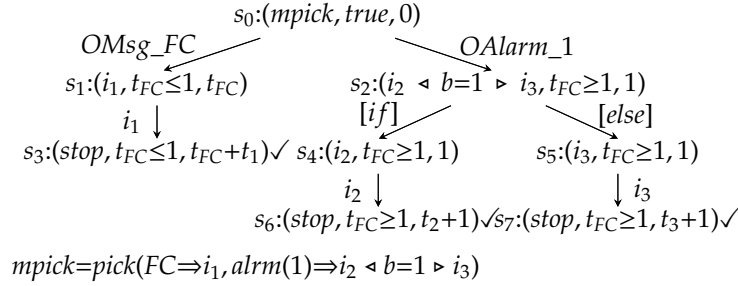
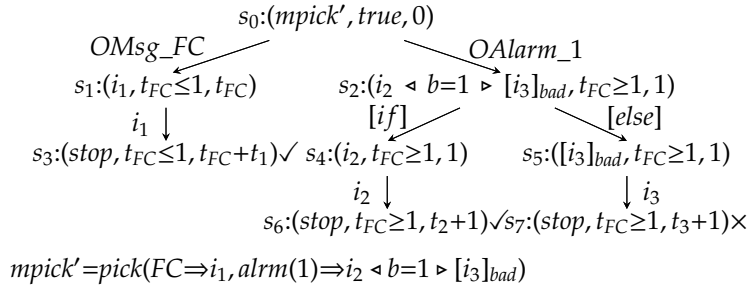
In this section, given a global time constraint  $T_G$  for a service  $\mathcal{M}$ , we present an approach to synthesize local time constraint  $C_L$  based on the LTS. We show that if response times of all component services of  $S$  satisfy the local time requirement, the service  $S$  would end at a good state regardless of the values input by the user.

### 6.4.1 Synthesis of Local Time Requirement

We assume a composite service  $\mathcal{M}$  and its LTS  $L_{\mathcal{M}} = (S, s_0, \Sigma, \rightarrow)$ ; let  $S_{good}$  be the set of all good states of service  $L_{\mathcal{M}}$ . In this section, we assume there is no bad state; we will discuss bad states in Section 6.4.2.

A critical problem is, given  $L_{\mathcal{M}}$ , to synthesize the local time requirement for service  $\mathcal{M}$ . We make two observations here. First, a good state  $s_g = (P_g, C_g, D_g) \in S_{good}$  is reachable from the initial state  $s_0$  iff  $C_g$  is satisfiable. Second, whenever the good state  $s_g$  is reached, we require that the total delay from initial state  $s_0$  to state  $s_g$  must not be larger than the global




 Figure 6.6: LTS of composite service  $S$ 

 Figure 6.7: LTS of composite service  $S'$ 

time constraint  $T_G$ , i.e.,  $D_g \leq T_G$ . To sum up, given a good state  $s_g = (P_g, C_g, D_g)$  where  $s_g \in S_{good}$ , we require the constraint  $(C_g \implies (D_g \leq T_G))$  to hold. The constraint means that whenever state  $s_g$  is reachable from initial state  $s_0$ , the total (parametric) delay from initial state  $s_0$  to state  $s_g$  must be less than the global time constraint  $T_G$ . The synthesized local time constraint for  $\mathcal{M}$  is the conjunction of such constraints for each good state  $s_g \in S_{good}$ , i.e.,  $\bigwedge_{(P_g, C_g, D_g) \in S_{good}} (C_g \implies (D_g \leq T_G))$ . We will show in Section 6.4.5 such that there does not exist a parameter valuation  $\pi$ , that can trivially satisfy the local time constraint by not satisfying the constraints of all good states, i.e.,  $\pi \not\models s_g.C_g$  for all  $s_g \in S_{good}$ .

Figure 6.6 shows the LTS of a composite service  $S$  that contains an *mpick* process, where  $i_j$  denotes  $sInv(s_j)$ , such that  $s_j$  is a component service with parametric response time  $t_j$ , for  $j \in \{1, 2, 3\}$ . For composite service  $S$  in Figure 6.6, we have three good states (states  $s_3$ ,  $s_6$ , and  $s_7$ ), and the synthesized local time requirement for composite service  $S$  is:

$$\begin{aligned} & ((t_{FC} \leq 1) \implies (t_{FC} + t_1 \leq 5)) \wedge \\ & ((t_{FC} \geq 1) \implies (t_2 + 1 \leq 5)) \wedge \\ & ((t_{FC} \geq 1) \implies (t_3 + 1 \leq 5)) \end{aligned}$$

which can be simplified as

$$\begin{aligned} & ((t_{FC} \leq 1) \implies (t_{FC} + t_1 \leq 5)) \wedge \\ & ((t_{FC} \geq 1) \implies (t_2 + 1 \leq 5) \wedge (t_3 + 1 \leq 5)) \end{aligned}$$

### 6.4.2 Addressing the Bad States

As an example, let us now change the definition of  $i_3$  in Figure 6.6 as  $[sInv(s_3)]_{bad}$ , resulting in the LTS shown in Figure 6.7, where state  $s'_7$  is a bad state. We use this example to provide the intuition how to modify the synthesized constraint to avoid reaching bad states. Note that the constraint  $s'_7.C = t_{FC} \geq 1$  is introduced by the *pick* activity. A way to avoid the reachability of  $s'_7$  is to prevent the transition  $OAlrm\_1$  from firing. An effective way to achieve this is by adding the inequality  $\neg s'_7.C$  to the synthesized constraint.

Therefore, given  $c_i = (s_i.C \implies (s_i.D \leq T_G))$ , for  $i \in \{3, 6\}$ , the local time requirement for composite service  $S'$  would be  $(c_3 \wedge c_6) \wedge \neg s'_7.C$ . This constraint can ensure the reachability of at least one of the good states and avoid the reachability of all bad states. (This will be proved in Section 6.4.5.)

### 6.4.3 Synthesis Algorithms

Algorithm 6.1 presents the entry algorithm for synthesizing the local time constraint for a given service  $CS$ , by traversing the  $LTS = (S, s_0, \Sigma, \rightarrow)$  of  $CS$ .

This algorithm makes use of a second algorithm  $synConsAOLTS(s)$  depicted in Algorithm 6.2. Given a state  $s = (P, C, D)$  in the LTS of service  $CS$ ,  $synConsAOLTS(s)$  returns a constraint  $c \in C_U$ . If state  $s$  is a good state (line 1), then it returns the constraint

---

**Algorithm 6.1:** Algorithm  $LocalTimeConstraint(s_0)$ 

---

**input** : Initial state  $s_0$   
**output**: The local time constraint  $C_L$

```
1  $Cons \leftarrow synConsAOLTS(s_0);$   
2 return  $Cons \wedge K_{bad};$ 
```

---

---

**Algorithm 6.2:** Algorithm  $synConsAOLTS(s)$ 

---

**input** : State  $s$  of LTS  
**output**: The constraint for LTS that starts at  $s$

```
1 if  $s$  is good state then  
2   return  $(s.C \implies (s.D \leq T_G));$   
3 if  $s$  is bad state then  
4    $K_{bad} = K_{bad} \wedge \neg(s.C);$   
5   return  $true;$   
6 if  $s$  is non-terminal state then  
7    $SC \leftarrow \{synConsAOLTS(s') \mid s' \in Enable(s)\};$   
8   return  $\bigwedge \{c \mid c \in SC\};$ 
```

---

$s.C \implies (s.D \leq T_G)$  (line 2), where  $T_G$  is the given global time constraint of the service  $CS$ .  $K_{bad} \in C_U$  is a static variable that is used to collect the negation of the constraint associated to states that are marked with a bad status. If state  $s$  is a bad state (line 3), then  $K_{bad}$  is conjuncted with negation of  $s.C$  (line 4), and returns  $true$  as the constraint (line 5). The reason for returning  $true$  is to ensure that that the returned constraint does not subsequently modify the constraint returned by  $synConsAOLTS(s_0)$ , since  $true \wedge C' = C'$ , for any  $C' \in C_U$ . If  $s$  is a non-terminal state (line 6),  $SC \in C_U$  is populated with the result of  $synConsAOLTS(s')$ , for each enabled state  $s'$  from non-terminal state  $s$  (line 7). Given  $A = \{c_1, \dots, c_n\} \subset C_U$ , we denote by  $\bigwedge A$  the conjunction of constraints in  $A$ , i.e.,  $c_1 \wedge \dots \wedge c_n$ . In line 8, the conjunction of all the elements in  $SC$  is returned as the constraint.

Let us now return to the description of Algorithm 6.1. Upon getting the constraint of  $Cons = synConsAOLTS(s_0)$  (line 1), the synthesized local time constraint of service  $CS$ , which is  $K_{bad} \wedge Cons$ , is returned as the final result (line 2).

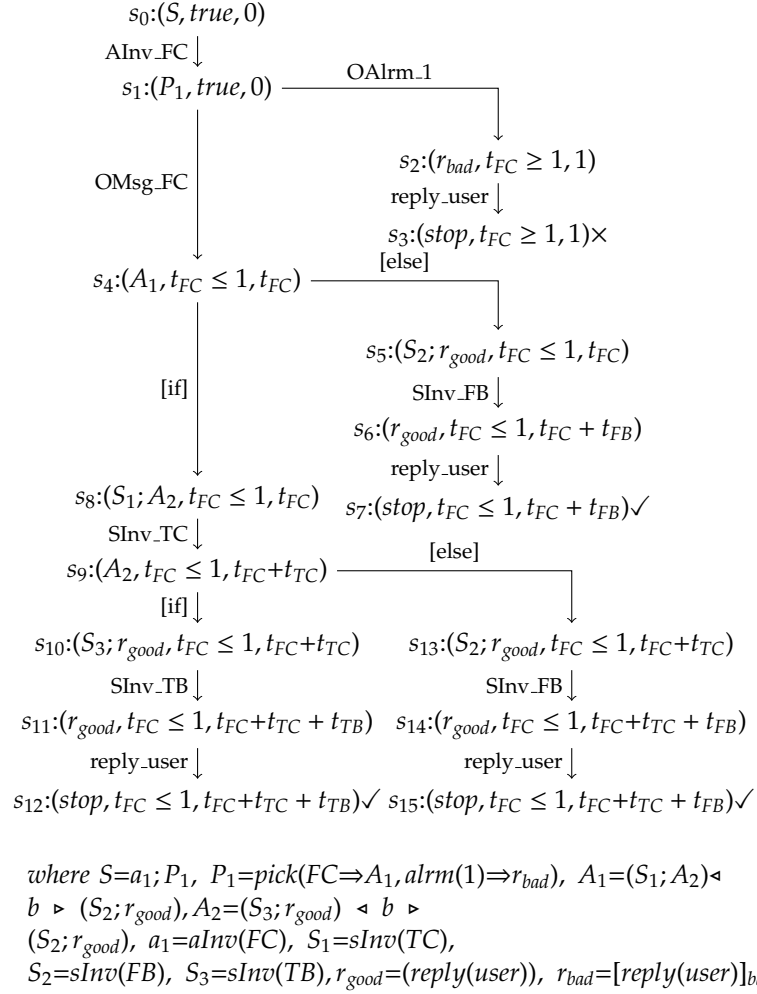


Figure 6.8: LTS of VBS

#### 6.4.4 Application to the Running Example

Figure 6.8 shows the LTS of the running example introduced in Section 6.1. Algorithm  $synConsAOLTS(s)$  is used to synthesize the local time requirement for VBS based on the LTS. The local time requirement of the running example is:

$$\begin{aligned} & \neg(t_{FC} \geq 1) \wedge \\ & (t_{FC} \leq 1) \implies (t_{FC} + t_{FB} \leq 5) \wedge \\ & (t_{FC} \leq 1) \implies (t_{FC} + t_{TC} + t_{TB} \leq 5) \wedge \\ & (t_{FC} \leq 1) \implies (t_{FC} + t_{TC} + t_{FB} \leq 5) \end{aligned}$$

and after simplification using Z3 [37], it becomes

$$\begin{aligned} & t_{FC} < 1 \wedge \\ & t_{FC} + t_{FB} \leq 5 \wedge t_{FC} + t_{TC} + t_{TB} \leq 5 \wedge t_{FC} + t_{TC} + t_{FB} \leq 5 \end{aligned}$$

This result provides us useful information regarding how the component services collectively satisfy the global time constraint. That is of most importance for selecting component services. For the case of *VBS*, one way to fulfill the global time requirement of *VBS* is to select service FC with response time that is less than 0.9 seconds, and all other services that have response times less than 2 seconds.

#### 6.4.5 Soundness

The algorithm is working on the abstract LTS to synthesize the local constraint at design time. On the other hand, the synthesized constraint is used in the concrete LTS during the runtime. Therefore, we use the abstract LTS for the properties that are concerned with the synthesis algorithm, while we use concrete LTS for the properties that are concerned with the usage of the synthesized constraint. In addition, we assume that all loops have a bound on the number of iterations and the execution time (see Section 5.3.3 for discussion).

**Lemma 6.4.1.** *Given  $\mathcal{M}$  be a service model, the abstract and concrete LTSs of  $\mathcal{M}$  are acyclic. **Proof.** This holds due to the assumption on the loop activities, such that the upper bound on the number of iterations and the time of execution, is known, and there are no recursive activities in BPEL.  $\square$*   
*end.*

Given a parameter valuation  $\pi$ , a state in a non-parametric service model  $\mathcal{M}[\pi]$  is said to be an intermediate (resp. final) state if it is a non-terminal (resp. terminal) state in parametric service model  $\mathcal{M}$ . A concrete LTS of  $\mathcal{M}[\pi]$  is *deadlockable* if and only if there exists an intermediate state  $s$  in the concrete LTS of  $\mathcal{M}[\pi]$  such that  $Enable(s) = \emptyset$ . This happens because  $\pi \not\models s'.C$ , for each  $s' \in Enable(s)$  in the concrete LTS of parametric service model  $\mathcal{M}$ . We show that  $\mathcal{M}[\pi]$  is not deadlockable in the next lemma.

**Lemma 6.4.2.** *Given a concrete service model  $\mathcal{M}$ , there do not exist a non-empty constraint  $C$  and a parameter valuation  $\pi \models C$ , such that the concrete LTS of  $\mathcal{M}[\pi]$  is deadlockable. **Proof.** Without loss of generality, assume the concrete LTS as  $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$ . The constraint of initial state is true, i.e.,  $s_0.C = true$ ; therefore it is always satisfiable. Given a state  $s$ , and a state  $s'$  such that  $s' \in Enable(s)$ , the situation where  $\pi \models s.C$  and  $\pi \not\models s'.C$  could only happen when if  $s'.C$  is stronger than  $s.C$ , i.e.,  $s'.C \subset s.C$ . In such a case, the additional constraints in  $s'.C$  must be introduced by pick or flow activity to constrain the relative speed of the services. Assume the pick construct as  $mpick = pick(S \Rightarrow P, alm(a) \Rightarrow Q)$ , where  $S$  is a service with parameter response time  $t_S$ ,  $a \in \mathbb{R}_{\geq 0}$  and  $P, Q$  are composite service activities. For the left branch of  $mpick$  to be enabled, the satisfaction of constraint  $t_S \leq a$  is required, while for right branch of  $mpick$  to be enabled, the satisfaction of constraint  $t_S \geq a$  is required. Since given any parameter valuation  $\pi$ ,  $mpick$  will be able to execute either of the branches, therefore it cannot be deadlocked. Assume the concurrent activity as  $conc = P \parallel Q$ , where  $P, Q$  are composite service activities. If  $P$  (resp.  $Q$ ) is a reply or asynchronous invocation activity, then  $P$  (resp.  $Q$ ) is always executable, since it takes no time. If  $P$  and  $Q$  are synchronous invocation activity or receive activity, which takes parameter response time  $t_P$  and  $t_Q$  respectively, then activity  $P$  is executable, if  $t_P \leq t_Q$ , and activity  $Q$  is executable if  $t_Q \leq t_P$ . Since given any parameter valuation  $\pi$ , either of the branches in  $conc$  is executable, therefore it cannot be deadlocked.  $\square$  **end.***

**Theorem 6.4.3** (Soundness). *Consider a service  $\mathcal{M}$  with its LTS  $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$ . Let  $S_{good}$  be the collection of good states in  $S$  and  $S_{bad}$  be the collection of all bad states in  $S$ . Let  $Q$  be the*

constraint synthesized by Algorithm 6.1 for service  $\mathcal{M}$  as  $Q$ . If  $Q$  is non-empty, given any parameter valuation  $\pi \models Q$ , it holds that:

1. At least one of the good states  $s_g \in S_{good}$  is reachable in any concrete LTS of  $\mathcal{M}[\pi]$ ;
2. There is no bad state  $s_b \in S_{bad}$  that is reachable in any concrete LTS of  $\mathcal{M}[\pi]$ .

**Proof.** To show item 1, we need to ensure that

1. The duration to reach any good state  $s_g \in S_{good}$  from initial state  $s_0$  satisfies the global time requirement;
2. At least one of the good states  $s_g \in S_{good}$  is reachable.

For item 1, the result is immediate. We therefore focus on item 2. The result of item 2 is followed from Lemma 6.4.1 since non-deadlock in non-terminal state implies that it will reach a terminal state which is either good state or bad state. Assume that it could not be a bad state (which we will show later), therefore it must be a good state, and item 2 holds.

To show item 2, we observe that  $K_{bad}$  contains the negated constraints from each of the bad states, i.e.,  $K_{bad} = \{\wedge(\neg s_b.C_i) | s_b \in S_{bad}\}$ . Since the synthesis result are conjuncted  $K_{bad}$ , therefore the constraints of all bad states are unsatisfiable, which implies that all bad states are non-reachable (this also completes the proof of item 2). In addition, the conjunction of  $K_{bad}$  in the synthesis result does not have the side effect of resulting in a deadlock in the concrete LTS of  $\mathcal{M}[\pi]$ , due to Lemma 6.4.2.  $\square$  **end.**

**Proposition 6.4.4.** Let  $\mathcal{M}$  be a concrete service model, and  $S_{good}$  be the set of all good states in concrete LTS of  $\mathcal{M}$ . Assume the synthesized constraint is  $Q_g \wedge K_{bad}$ , where  $Q_g = \wedge_{(P_i, C_i, D_i) \in S_{good}} (C_i \implies (D_i \leq T_G))$ , and  $T_G$  be the global time constraint. Given the constraint  $Q_g$  as non-empty, there does not exist a parameter valuation  $\pi$  such that  $\pi \models Q_g$  and  $\pi \not\models s_g.C_i$  for all  $s_g \in S_{good}$ . **Proof.**

Assume there exists such a parameter valuation  $\pi$ . According to Theorem 6.4.3, at least a good state is reachable in concrete LTS of  $\mathcal{M}[\pi]$ . Without loss of generality, assume a good state that is reachable as  $(P, C, D)$ , which implies that  $\pi \models C$ . This contradicts the assumption.  $\square$  *end*.

## 6.5 Evaluation

In this section, we report about the evaluation of our approach using two case studies. Each case study consists of a service composition in the form of a BPEL process. The experiment data were obtained on a system using Intel Core I5 2410M CPU with 4 GB RAM.

### 6.5.1 Stock Market Indices Service

The Stock marked indices service (in short SMIS) is used as a running example in [88]. The goal of SMIS is to provide updated stock indexes to the subscribed users. It provides a service level agreement (SLA) to the subscribed users stating that it always responds within three seconds upon request. The SMIS has three component Web services: a database service (*DS*), a free news feed service (*FS*) and a paid news feed service (*PS*). The strategy of the SMIS is calling the free service *FS* before calling the paid service *PS* in order to minimize the cost. Upon returning the result to the user, the SMIS would also store the latest results in an external database service provided by *DS* (storage of the results is omitted here). Upon receiving the response from *DS*, the process is followed by an `<if>` branch. If the indexes are available, then they are returned to the user. Otherwise, *FS* is invoked asynchronously. A `<pick>` construct is used here to await incoming response from previous asynchronous invocation and timeout if necessary. If the response from *FS* is received within one second, then the result is returned to the user. Otherwise, the timeout occurs, and SMIS stops waiting for the result from *FS* and calls *PS* instead. Similar to *FS*, the result from *PS* is returned to user, if the response from *PS* is received within one second.



Otherwise, it would notify the user regarding the failure of getting stock indexes. The resulting LTS has 15 states with 14 transitions, and it takes 0.0076 seconds for synthesizing the gLTC, and it takes 0.0078 seconds for synthesizing the sLTC for each state of the LTS. The gLTC for TBS after simplification is shown below.

$$\begin{aligned}
 & ((t_{FS} < 1) \wedge ((t_{DS} + t_{FS}) \leq 3) \wedge (t_{DS} \leq 3)) \\
 \vee & ((t_{FS} < 1) \wedge (t_{DS} \leq 2) \wedge ((t_{DS} + t_{PS}) \leq 2) \wedge (t_{PS} \leq 2)) \\
 \vee & ((t_{PS} < 1) \wedge (t_{FS} > 1) \wedge ((t_{DS} + t_{PS}) \leq 2) \wedge (t_{DS} \leq 2)) \\
 \vee & ((t_{PS} < 1) \wedge ((t_{DS} + t_{FS}) \leq 3) \wedge (t_{DS} \leq 2) \wedge (t_{FS} \leq 3) \wedge ((t_{DS} + t_{PS}) \leq 2))
 \end{aligned}$$

### 6.5.2 Computer Purchasing Services

We consider in this section a computer purchasing service (CPS). The goal of a CPS (e.g., Dell.com) is to allow a user to purchase the computer system online using credit cards. Our CPS makes use of five component services, namely Shipper Service (SS), Logistic Service (LS), Inventory Service (IS), Manufacture Service (MS), and Billing Service (BS). The global time requirement of the CPS is to response within three seconds. The CPS starts upon receiving the purchase request from the client with credit card information, and the CPS spawns three workflows (viz., shipping workflow, inventory workflow, and billing workflow) concurrently. In the shipping workflow, the shipping service provider is invoked synchronously for the shipping service on computer systems. Upon receiving the reply, LS which is a service provided by internal logistic department is invoked synchronously to record the shipping schedule. In the manufacture workflow, IS is invoked synchronously to check for the availability of the goods. Subsequently, MS is invoked asynchronously to update the manufacture department regarding the current inventory stock. In the billing workflow, the billing service which is offered by third party merchant, is invoked synchronously for billing the customer with credit card information. The LTS of this system contains 121 states and 120 transitions. The time taken for the synthesis gLTC takes 0.0532

seconds, and the time taken for the synthesis sLTC for each state of the LTS takes 0.0562 seconds. The gLTC for CPS is given below.

$$\boxed{((t_{SS} + t_{LS} + t_{IS} + t_{BS}) \leq 3) \wedge (t_{SS} \leq 3) \wedge (t_{LS} \leq 3) \wedge (t_{IS} \leq 3) \wedge (t_{BS} \leq 3)}$$

Note that  $t_{MS}$  does not appear in the local time constraint for CPS. The reason is that MS is invoked asynchronously without expecting a response; therefore its response time is irrelevant to the global time requirement of CPS.

### 6.5.3 Travel Booking Service

The goal of a travel booking service (TBS) (such as Booking.com) is to provide a combined flight and hotel booking service by integrating two independent existing services. TBS provides an SLA for its subscribed users, saying that it must respond within five seconds upon request. The travel booking system has five component services, user validation service (VS), flight service (FS), backup flight service ( $FS_{bak}$ ), hotel service (HS) and backup hotel service ( $HS_{bak}$ ). Upon receiving the request from users, TBS spawns two workflows (viz., a flight request workflow, and a hotel request workflow) concurrently. In the flight request workflow, it starts by invoking FS, which is a service provided by a flight service booking agent. If service FS does not respond within two seconds, then FS is abandoned, and another backup flight service  $FS_{bak}$  is invoked. If  $FS_{bak}$  returns within one second, then the workflow is completed; otherwise it is considered as a failure for the flight request workflow. The hotel request workflow shares the same process as the flight request workflow, by replacing FS with HS and  $FS_{bak}$  with  $HS_{bak}$ . The resulting LTS has 562 states with 2387 transitions, and it takes 1.004 seconds for synthesizing the LTC and it takes 1.05 seconds for synthesizing the sLTC for each state in LTS respectively. The local time constraint for TBS is shown below.

$$\begin{aligned}
& ((t_{HSbak} < 1) \wedge (t_{FSbak} < 1) \wedge ((t_{FSbak} + t_{HSbak}) \leq 1)) \\
\vee & ((t_{HSbak} < 1) \wedge (t_{FS} < 2)) \\
\vee & ((t_{HS} < 2) \wedge (t_{FSbak} < 1)) \\
\vee & ((t_{HS} < 2) \wedge (t_{FS} < 2)) \\
\vee & ((t_{FSbak} > 2t_{HSbak}) \wedge (t_{HSbak} > 2t_{FSbak})) \\
\wedge & (t_{HSbak} < 1) \wedge (t_{FSbak} < 1)
\end{aligned}$$

## 6.6 Related Work

This work shares common techniques with work for constraint synthesis for scheduling problems. The use of models such as Parametric Timed Automata (PTA) [14] and Parametric Time Petri Nets (TPNs) [91] for solving such problems has received recent attention. In particular, in [30, 66, 47], parametric constraints are inferred, guaranteeing the feasibility of a schedule using PTAs with stopwatches. In [15], we extended the “inverse method” (see, e.g., [16]) to the synthesis of parameters in a parametric, timed extension of CSP. Although PTAs or TPNs might have been used to encode (part of) BPEL language, our work is specifically adapted and optimized for synthesizing local timing constraint in the area of service composition. The quantitative measure of the robustness of concurrent timed systems has been tackled in different papers (see [69] for a survey). However, most approaches consider a single dimension  $\epsilon$ : transitions can usually be taken at most  $\epsilon$  (before or after) units of time from their original firing time. This can be seen as a “ball” in  $|U|$  dimensions of radius  $\epsilon$ . In contrast, our approach quantifies robustness for all parameter dimensions, in the form of a polyhedron in  $|U|$  dimensions.

Our method is related to using LTS for analysis purpose in Web services. In [24], the author proposes an approach to obtain behavioral interfaces in the form of LTS of external services by decomposing the global interface specification. It also has been used in the model checking the safety and liveness properties of BPEL services. For example, Foster

et al. [42, 45] transform BPEL process into FSP, subsequently using a tool named as WS-Engineer for checking safety and liveness properties. Simmonds et al. [81] proposes a user-guided recovery framework for Web services based on LTS. Our work uses LTS in synthesizing local time requirement dynamically.

Our method is related to the finding of a suitable quality of service (QoS) for the system [97]. The authors of [97] propose two models for the QoS-based service composition problem [17] model the service composition problem as a mixed integer linear problem where constraints of global and local component serviced can be specified. The difference with our work is that, in their work, the local constraint has been specified, whereas for ours, the local constraints is to be synthesized. An approach of decomposing the global QoS to local QoS has been proposed in [8]. It uses the mixed integer programming (MIP) to find optimal decomposition of QoS constraint. However, the approach only concerns for simplistic sequential composition of Web services method call, without considering complex control flow and timing requirement.

Our method is related to response time estimation. In [63], the authors propose to use linear regression method and a maximum likelihood technique for estimating the service demands of requests based on their response times. [70] has also discussed the impact of slow services on the overall response time on a transaction that use several services concurrently. Our work is focused on decomposing the global requirement to local requirement, which is orthogonal to these works.



## Chapter 7

# Conclusion

This chapter summarizes this thesis. We will conclude the thesis in Section 7.1 and discuss the on-going work and future directions in Section 7.2.

### 7.1 Summary

In this thesis, we study the verification and analysis of web service composition. In the following, we briefly summarize our contributions of the thesis again.

Firstly, we have proposed practical solutions to link two different views (choreography and orchestration) of Web services using model checking methods. We propose formal languages for modeling choreography and orchestration respectively with formal operational semantics, which create a unified semantics model for the two views, so that it allows communications between choreography and orchestration models. In addition, we propose a method to mechanically synthesize a prototype a Web service orchestration from the choreography, by repairing the choreography if necessary and projecting relevant behaviors to each service provider.

Secondly, we provide functional verification for Orc programs. Orc is a hierarchical concurrent language that has highly concurrent semantics, and this has posed a challenge of state-explosion problem for its verification. To address this problem, we also proposed a new method, called Compositional Partial Order Reduction (CPOR), which aims to provide the reduction with a greater scale than current partial order reduction methods in the context of hierarchical concurrent processes. CPOR exploits the independency within local transitions. It applies POR recursively for the hierarchical concurrent processes, and several possible ample sets are composed in a bottom-up manner. It has been used in model checking Orc programs to verify the functional requirements of Web service composition. Experiment results show that CPOR provides significant state-reduction for Orc programs.

Thirdly, we have illustrated our approach to verify combined functional and non-functional requirements (i.e., availability, response time and cost) for Web service composition. To the best of our knowledge, we are the first work on such integration. We capture the semantics of web service composition using labeled transition systems (LTSs) and verify the Web service composition directly without building intermediate or abstract models before applying verification approaches. For different kinds of non-functional requirements, we have proposed different aggregation functions. Furthermore, our experiment shows that our approach can work on real-world BPEL programs efficiently.

Lastly, given the global time requirement, we propose an automated method for synthesizing the local time requirement for component services of a composite service. Our approach is based on the dynamic analysis of the LTS of a composite service by making use of parameterized timed techniques.

## 7.2 Ongoing and Future Work

In Chapter 3 we have illustrated a model-based method for fully automatic analysis of Web service composition, which offered a lightweight approach to tackle the synthesis problem. In future work, we plan to investigate state reduction methods that can be used to increase the efficiency of conformance checking.

In Chapter 4, we have presented the verification of *Orc* program, and have proposed a state reduction method, called Compositional Partial Order Reduction (CPOR), to provide state-reduction for the verification of *Orc* program. As for future works, we would further evaluate CPOR by applying it for verifying programs in other languages that has hierarchical concurrent structure, such as CSP [52].

In Chapter 5, we have presented that our approach to verify combined functional and non-functional requirements for Web service composition can work on real-world BPEL programs efficiently. For future directions, we will consider various heuristics that could be used to reduce the number of states and transitions for effective verification. Another possible direction is to extend this work to other domains that share similar problems such as sensor network [98].

In Chapter 6, we have evaluated our technique for synthesizing the local time requirement for the component services of a composite service with real-world case studies. However, it is just the starting of this work. We plan to investigate the combination of our approach with other approaches such as the "inverse method" [15] to evaluate the possibility of synthesizing a better local time requirement.





# Bibliography

- [1] OASIS Standards. <http://www.oasis-open.org/standards>. 2.1.1
- [2] PAT: Process Analysis Toolkit. <http://www.comp.nus.edu.sg/~pat/research/>. 4.3
- [3] Web Services Glossary. <http://www.w3.org/TR/ws-gloss/>. 2.1.1
- [4] World WideWeb Consortium. Extensible markup language (XML). <http://www.w3c.org/XML>. 2.1.1
- [5] Simple Object Access Protocol (SOAP) 1.1. Technical report, May 2000. <http://www.w3.org/TR/SOAP/>. 2.1.1, 5
- [6] Web Services Description Language (WSDL) 1.1. Technical report, March 2001. <http://www.w3.org/TR/wsdl>. 2.1.1, 5
- [7] W. M. P. v. d. Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance Checking of Service Behavior. *ACM Trans. Internet Technol.*, 8(3):1–30, 2008. 3.5
- [8] M. Alrifai and T. Risse. Combining Global Optimization with Local Selection for Efficient QoS-Aware Service Composition. In *WWW 2009*, pages 881–890. ACM, 2009. 6.6
- [9] M. AlTurki and J. Meseguer. Real-time Rewriting Semantics of Orc. In *PPDP*, pages 131–142, 2007. 4.3, 4.4
- [10] M. AlTurki and J. Meseguer. Reduction Semantics and Formal Analysis of Orc Programs. *Electr. Notes Theor. Comput. Sci.*, 200(3):25–41, 2008. 4.3, 4.4
- [11] M. AlTurki and J. Meseguer. Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis. Technical report, The University of Illinois at Urbana-Champaign, April 2010. <https://www.ideals.illinois.edu/handle/2142/15414>. 4.3
- [12] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical computer science*, 126(2):183–235, 1994. 6.2.1, 6.3
- [13] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003. 3, 3.3

- [14] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric Real-time Reasoning. In *STOC 1993*, pages 592–601. ACM, 1993. 6.2.1, 6.6
- [15] É. André, Y. Liu, J. Sun, and J. S. Dong. Parameter Synthesis for Hierarchical Concurrent Real-Time Systems. In *ICECCS*, pages 253–262, 2012. 6.3, 6.6, 7.2
- [16] É. André and R. Soulat. *The Inverse Method*. ISTE Ltd and John Wiley & Sons Inc., 2013. 6.6
- [17] D. Ardagna and B. Pernici. Global and Local QoS Guarantee in Web Service Selection. In *Business Process Management Workshops*, 2005. 6.6
- [18] C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, 2007. 2.2, 4, 4.2.2, 4.2, 5
- [19] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step. In *WS-FM'05*, pages 257–271, 2005. 3.5
- [20] T. Basten and D. Bosnacki. Enhancing Partial-Order Reduction via Process Clustering. In *ASE*, pages 245–253, 2001. 4.3, 4.4
- [21] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. 6.3
- [22] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-Services. In *FSE'09*, pages 141–150, 2009. 3.5
- [23] D. Bianculli, C. Ghezzi, and P. Spoletini. A Model Checking Approach to Verify BPEL4WS Workflows. In *SOCA'07*, pages 13–20, 2007. 3.5
- [24] D. Bianculli, D. Giannakopoulou, and C. S. Pasareanu. Interface decomposition for service compositions. In *ICSE*, pages 501–510, 2011. 6.6
- [25] Y. Bontemps and P. Schobbens. The Complexity of Live Sequence Charts. In *FOSSACS'05*, pages 364–378, 2005. 3.3
- [26] T. Bultan and X. Fu. Specification of Realizable Service Conversations Using Collaboration Diagrams. In *SOCA'07*, pages 122–132, 2007. 3.5
- [27] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. *WCD-Working Note*, 2006. <http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf>. 2.1.2.1, 3, 3.1, 3.3, 3.4, 3.5
- [28] B. F. Chellas. *Modal Logic: an Introduction*, volume 980. Cambridge Univ Press, 1979. 2.3
- [29] M. Chen, T. H. Tan, J. Sun, Y. Liu, and J. S. Dong. Verification of Functional and Non-functional Requirements for Web Service Composition. In *ICFEM*, 2013. 1.4

- [30] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In *RTSS 2008*, pages 80–89. IEEE Computer Society, 2008. 6.6
- [31] E. M. Clarke and E. A. Emerson. *Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic*. Springer, 1982. 2.4.2
- [32] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. 2.4.2
- [33] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. In *CAV*, pages 450–462. Springer, 1993. 4
- [34] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. The MIT press, 1999. 2.2, 4.1.2
- [35] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992. 2.4.2, 5.3.1
- [36] P. J. Courtois, F. Heymans, and D. Parnas. Concurrent Control with “Readers” and “Writers”. *Commun. ACM*, 14(10):667–668, 1971. 4.3
- [37] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS 2008*, LNCS, pages 337–340. Springer, 2008. 6.4.4
- [38] E. W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. 1
- [39] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *ICFEM’06*, pages 226–245. Springer, 2006. 3.5, 4.4
- [40] E. A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. In *CADE’00*, pages 236–254, London, UK, 2000. Springer-Verlag. 3.2
- [41] E. A. Emerson and A. P. Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997. 4
- [42] H. Foster. *A Rigorous Approach To Engineering Web Service Compositions*. PhD thesis, Imperial College of London, 2006. 6.6
- [43] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model Checking Service Compositions under Resource Constraints. In *FSE’07*, pages 225–234, 2007. 3, 3.1.2, 3.4, 3.5
- [44] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *ASE’03*, pages 152–163, 2003. 3.5

- [45] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a Tool for Model-Based Verification of Web Service Compositions and Choreography. In *ICSE'06*, pages 771–774, 2006. 3.5, 6.6
- [46] H. Foster, S. Uchitel, J. Magee, and J. Kramer. WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography. In *Test and Analysis of Web Services*, pages 87–119. 2007. 5, 5.5
- [47] L. Fribourg, D. Lesens, P. Moro, and R. Soulat. Robustness Analysis for Scheduling Problems using the Inverse Method. In *TIME 2012*, pages 73–80. IEEE Computer Society Press, 2012. 6.6
- [48] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *WWW'04*, pages 621–630, 2004. 3.5
- [49] C. K. Fung, P. C. K. Hung, G. Wang, R. C. Linger, and G. H. Walton. A Study of Service Composition with QoS Management. In *ICWS '05*, pages 717–724, 2005. 5, 5.5
- [50] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. 4, 4.2
- [51] J. Håkansson and P. Pettersson. Partial Order Reduction for Verification of Real-Time Components. In *FORMATS*, pages 211–226, 2007. 4.2
- [52] C. A. R. Hoare. *Communicating Sequential Processes*. International Series on Computer Science. Prentice-Hall, 1985. 3.1.1, 7.2
- [53] G. J. Holzmann. On-The-Fly Model Checking. *ACM Comput. Surv.*, 28(4es):120, 1996. 4
- [54] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. 2.4.2, 3.4
- [55] C. N. Ip and D. L. Dill. Verifying Systems with Replicated Components in Murphi. In *CAV'96*, pages 147–158, 1996. 3.2
- [56] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0. <http://www.oasis-open.org/specs/#wsbpelv2.0>, Apr 2007. 1.1, 2.1.2.1, 3, 5, 6, 6.1
- [57] R. Kazhamiakin, P. K. Pandya, and M. Pistore. Representation, Verification, and Computation of Timed Properties in Web. In *ICWS'06*, pages 497–504, 2006. 3.5
- [58] R. Kazhamiakin and M. Pistore. Choreography Conformance Analysis: Asynchronous Communications and Information Alignment. In *WS-FM'06*, pages 227–241, 2006. 3.5
- [59] D. Kitchin. Operational and Denotational Semantics of the Otherwise Combinator. <http://orc.csres.utexas.edu/papers/OrcOtherwiseSemantics.pdf>, 2009. 4, 4.1.2
- [60] D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc Programming Language. In *FMOODS/FORTE*, pages 1–25, 2009. 1.1, 2.1.2.1, 3.1, 4, 4.1.1, 4.1.1

- [61] D. Kitchin, A. Quark, and J. Misra. Quicksort: Combining Concurrency, Recursion, and Mutable Data Structures. Technical report, The University of Texas at Austin, Department of Computer Sciences. 4.3
- [62] S. Koizumi and K. Koyama. Workload-aware Business Process Simulation with Statistical Service Analysis and Timed Petri Net. In *ICWS '07*, pages 70–77. IEEE CS, 2007. 5, 5.5
- [63] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating Service Resource Consumption from Response Time Measurements. In *VALUETOOLS*, page 48, 2009. 6.6
- [64] J. P. Krimm and L. Mounier. Compositional State Space Generation with Partial Order Reductions for Asynchronous Communicating Systems. In *TACAS*, pages 266–282, 2000. 4.3, 4.4
- [65] F. Lang and R. Mateescu. Partial Order Reductions Using Compositional Confluence Detection. In *FM*, pages 157–172, 2009. 4.3, 4.4
- [66] T. T. H. Le, L. Palopoli, R. Passerone, Y. Ramadian, and A. Cimatti. Parametric Analysis of Distributed Firm Real-time Systems: A Case Study. In *ETFA 2010*, pages 1–8, 2010. 6.6
- [67] B. Li, Y. Zhou, and J. Pang. Model-Driven Automatic Generation of Verified BPEL Code for Web Service Composition. In *APSEC'09*, pages 355–362. IEEE CS, 2009. 5.5
- [68] Y. Liu. *Model Checking Concurrent and Real-time Systems: the PAT Approach*. PhD thesis, National University of Singapore, 2010. 4.3
- [69] N. Markey. Robustness in Real-time Systems. In *SIES 2011*, pages 28–34. IEEE, 2011. 6.6
- [70] D. A. Menascé. Response-Time Analysis of Composite Web Services. *IEEE Internet Computing*, 8(1):90–92, 2004. 6.6
- [71] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In *WWW'08*, pages 815–824. ACM, 2008. 5.3.3
- [72] S. Nakajima. Lightweight formal analysis of web service flows. *Progress in Informatics*, 2:57–76, 2005. 5, 5.5
- [73] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *CAV*, pages 377–390, 1994. 4
- [74] D. Peled. Ten Years of Partial Order Reduction. In *CAV*, pages 17–28, 1998. 4
- [75] A. Pnueli and R. Rosner. Distributed Reactive Systems are Hard to Synthesis. In *FOCS'90*, pages 746–757, 1990. 3
- [76] G. Pu, J. Shi, Z. Wang, L. Jin, J. Liu, and J. He. The Validation and Verification of WSCDL. In *APSEC'07*, pages 81–88. IEEE Computer Society, 2007. 3.1, 3.5
- [77] Y. Qian, Y. Xu, Z. Wang, G. Pu, H. Zhu, and C. Cai. Tool Support for BPEL Verification in ActiveBPEL Engine. In *ASWEC '07*, pages 90–100, 2007. 5.5

- [78] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the Theoretical Foundation of Choreography. In *WWW'07*, pages 973–982, 2007. 3.1, 3.4, 3.5
- [79] A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994. 1.1, 3, 3.2, 3.4
- [80] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1986. 6.3.6
- [81] J. Simmonds, S. Ben-David, and M. Chechik. Guided Recovery for Web Service Applications. In *SIGSOFT FSE 2010*, pages 247–256, 2010. 6.6
- [82] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985. 2.4.2, 4.3
- [83] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and É. André. Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology*, 22(1):3.1–3.29, feb 2013. 6.3
- [84] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009. 1.1, 3.2, 5.3.1
- [85] J. Sun, Y. Liu, J. S. Dong, G. Pu, and T. H. Tan. Model-based Methods for Linking Web Service Choreography and Orchestration. In *APSEC 2010*, pages 166 – 175, 2010. 1.4
- [86] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In A. Cavalcanti and D. Dams, editors, *Proceedings of the Second World Congress on Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2009. 4.3
- [87] T. H. Tan. Towards verification of a service orchestration language. In *SSIRI (Companion)*, pages 36–37, 2010. 4.4
- [88] T. H. Tan, É. André, J. Sun, Y. Liu, J. S. Dong, and M. Chen. Dynamic synthesis of local time requirement for service composition. In *ICSE*, pages 542–551, 2013. 1.4, 5.5, 6.5.1
- [89] T. H. Tan, Y. Liu, J. Sun, and J. S. Dong. Compositional Partial Order Reduction for Model Checking Concurrent Systems. Technical report, National Univ. of Singapore, August 2010. <http://www.comp.nus.edu.sg/~pat/fm/cpor/CPORTR.pdf>. 4.1.3, 4.3
- [90] T. H. Tan, Y. Liu, J. Sun, and J. S. Dong. Verification of Orchestration Systems Using Compositional Partial Order Reduction. In *ICFEM*, pages 98–114, 2011. 1.4
- [91] L.-M. Traonouez, D. Lime, and O. H. Roux. Parametric Model-Checking of Stopwatch Petri Nets. *Journal of Universal Computer Science*, 15(17):3273–3304, 2009. 6.6
- [92] A. Valmari. The State Explosion Problem. In *Petri Nets*, pages 429–528, 1996. 4

- [93] I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. A Timed Semantics of Orc. *Theoretical Computer Science*, 402(2):234–248, 2008. 2.1.2.1, 4, 4.1.2, 4.1.2, 4.1.4, 4.2.3
- [94] H. Xiao, B. Chan, Y. Zou, J. W. Benayon, B. O’Farrell, E. Litani, and J. Hawkins. A Framework for Verifying SLA Compliance in Composed Services. In *ICWS ’08*, pages 457–464, 2008. 5.5
- [95] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do Fixes Become Bugs? *ESEC/FSE ’11*, pages 26–36, 2011. 1, 5
- [96] J. Yu, T. P. Manh, J. Han, Y. Jin, Y. Han, and J. Wang. Pattern Based Property Specification and Verification for Service Composition. In *WISE’06*, pages 156–168, 2006. 5.5
- [97] T. Yu, Y. Zhang, and K.-J. Lin. Efficient Algorithms for Web Services Selection with End-to-end QoS Constraints. *TWEB*, 1(1), 2007. 6.6
- [98] M. Zheng, J. Sun, D. Sanán, Y. Liu, J. S. Dong, and Y. Gu. Towards bug-free implementation for wireless sensor networks. In *SenSys*, pages 407–408, 2011. 7.2